

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

Dissertation zur Erlangung des Doktorgrades der Naturwissenschaften (Dr. rer. nat.)

der

Naturwissenschaftlichen Fakultät I – Biowissenschaften –
der Martin-Luther-Universität Halle-Wittenberg,

Vorgelegt von

Herr Sam C. Levin, M.Sc.

geboren am 23.01.1990 in Philadelphia, PA, USA

Gutachter

1. Professor Tiffany Knight (Martin Luther University Halle-Wittenberg, Helmholtz Center for Environmental Research, German Centre for Integrative Biodiversity)
- 2.
- 3.

Datum der Verteidigung: 18.01.2022

Contents

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

Summary	2
Chapter 1: Introduction	2
Chapter 2: Flexible implementation of Integral Projection Models in R	4
Chapter 3: PADRINO and Rpadrino: a toolkit for rebuilding and analyzing published Integral Projection Models	14
How climate change will alter the threat of invasive species: a case study with the <i>Carpobrotus</i> genus	27
Synthesis	28
Acknowledgements	29
Appendix 1: ipmr Case Study 1	29
Appendix 2: ipmr Case Study 2	48
Supplementary Information for Chapter 2	59
Appendix 4: PADRINO Case Study 1	60
Appendix 5: PADRINO Case Study 2	97
Appendix 6: Supplementary Information for Chapter 2	112
Eigenständigkeitserklärung	113

Summary

The study of structured populations and their dynamics has become a key aspect of many sub-disciplines within ecology as a whole. Currently, matrix population models (MPMs) are the most widely used structured population models. Integral projection models (IPMs) are more new, but are rising in popularity. A variety of tools and techniques to parameterize and implement IPMs have been concurrently created. However, pressing questions in ecology, evolution, and conservation, and their associated data and computational requirements, have outpaced the development of tooling to aid their implementation, interpretation, and synthesis. This dissertation aims to partially fill this gap by introducing new computational tools and data collection techniques. These tools and techniques enable large scale data collection, ease the implementation of complex IPMs, and provide a more streamlined experience for syntheses.

In **Chapter 2**, I introduce a new *R* package, `ipmr`, to implement and analyse IPMs. This *R* package provides a domain specific language to implement deterministic and stochastic models, with two options for handling the latter. Furthermore, it handles density dependence with minimal additional overhead. It is designed to accommodate almost any type of regression model used in the vital rate fitting process, and also includes methods for extracting the basic building blocks used in many downstream analyses. This chapter includes two appendices with detailed case studies demonstrating the package's use, and the package itself contains six vignettes that are intended for audiences from a range of technical backgrounds.

In **Chapter 3**, I introduce two additional tools: the PADRINO IPM Database, and `Rpadrino`, an *R* package for interfacing with PADRINO. PADRINO houses IPMs extracted from published peer-reviewed studies, as well as extensive metadata that enables researchers to filter the database down to models that fit their research questions. `Rpadrino` provides a clean interface to rebuild these models from *R* using the `ipmr` *R* package as a backend. This chapter also includes two detailed case studies on how to use these tools for both standalone analyses and in conjunction with other databases to power much larger scale syntheses.

In **Chapter 4**, I examine the mechanisms underlying invasions by the *Carpobrotus* genus using data collected by drones on four continents. We show that native status and genetic composition matter MORE/LESS/EQUALLY 0 than climate in driving the success of this widely invasive genus. Furthermore, this chapter demonstrates the utility of new demographic data collection methods for at least some environmental contexts.

Chapter 1: Introduction

Integral projection models: past and present

There is a rich history of dividing populations into sub-groups to better understand how variation within a population affects the past, present, and future of the population as a whole. Generally, Alfred Lotka's work in the early 1900s to understand age-specific rates of birth and death is recognized as the beginning of this field of study (though work going as far back as the late 1600s by John Graunt and Edmund Halley do contain elements of this underlying idea). However, this approach did not become popular in ecology for some time, likely due to the tedious nature of the calculations (Ebert 1999). P.H. Leslie introduced the age structured projection matrix in 1945. This approach provided a more elegant way of summarizing vital rates, though still required dutiful attention to detail when performing the calculations by hand. Extensions of Leslie's age-based approach to incorporate stage-structured populations (Lefkovich 1965) and multiple structuring variables (Goodman 1969) advanced theory, but did not necessarily drive broader uptake. The advent of personal computers and the recognition that numerical analysis methods fit well with this rapidly growing technology lead to a rapid adoption in the late 1970s and early 1980s. Concurrently, Hal Caswell and colleagues' showed how to calculate on sensitivity (1978) and elasticity (1986) of population growth rates, which are still a vital analytical tool for applied practitioners and theoreticians. A subsequent text book provided computer code to implement these analyses, along with many others, helped move these approaches from the esoteric to the practical (Caswell 1989).


Somewhere along the way, researchers realized they may not be able to sub-divide all populations neatly. Many species have their life cycle structured by some continuous trait, like weight, height, diameter, or hatching date. The integral projection model (IPM) was proposed to alleviate this issue (Easterling et al. 2000). IPM theory provides a framework to model the dynamics of populations structured by any number of continuous and discrete traits (Ellner & Rees 2006, Ellner et al. 2016).

Citations

1. Caswell, H. (1978). A general formula for the sensitivity of population growth rate to changes in life history parameters. *Theoretical Population Biology* 14(2): 215-230. <https://www.jstor.org/stable/2528566>
2. Easterling, M.R., Ellner, S.P., & Dixon, P.M. (2000). Size-specific sensitivity: Applying a new structured population model. *Ecology* 81(3): 694-708. [https://doi.org/10.1890/0012-9658\(2000\)081%5B0694:SSSAAN%5D2.0.CO;2](https://doi.org/10.1890/0012-9658(2000)081%5B0694:SSSAAN%5D2.0.CO;2)
3. Ebert, T.A. (1999). *Plant and Animal Populations: Methods in Demography*. Academic Press, San Diego, California.
4. Ellner, S.P. & Rees, M. (2006) *General IPMs*
5. Ellner, S.P., Childs, D.Z., & Rees, M. (2016) *Data driven modelling of structured populations: a practical guide to the Integral Projection Model*. Springer, Switzerland.
6. de Kroon, H., Plaisier A., van Groenendael, J., & Caswell, H. (1986). Elasticity: the relative contribution of demographic parameters to population growth rate. *Ecology* 67(5): 1427-1431. <https://doi.org/10.2307/1938700>.
7. Lefkovich, L.P. (1965). The study of population growth in organisms grouped by stages. *Biometrics* 21(1): 1-18. <https://www.jstor.org/stable/2528348>

Chapter 2: Flexible implementation of Integral Projection Models in R

ipmr: Flexible implementation of Integral Projection Models in R

Sam C. Levin^{1,2,3}  | Dylan Z. Childs⁴  | Aldo Compagnoni^{1,2,3}  | Sanne Evers^{1,2,5}  |
Tiffany M. Knight^{1,2,5}  | Roberto Salguero-Gómez³ 

¹Institute of Biology, Martin Luther University Halle-Wittenberg, Halle (Saale), Germany

²German Centre for Integrative Biodiversity Research (iDiv) Halle-Jena-Leipzig, Leipzig, Germany

³Department of Zoology, University of Oxford, Oxford, UK

⁴Department of Animal and Plant Sciences, University of Sheffield, Sheffield, UK

⁵Department of Community Ecology, Helmholtz Centre for Environmental Research-UFZ, Halle (Saale), Germany

Correspondence

Sam C. Levin

Email: levisc8@gmail.com

Funding information

R.S.-G. was supported by a NERC Independent Research Fellowship (NE/M018458/1). S.C.L., A.C., S.E. and T.M.K. were funded by the Alexander von Humboldt Foundation in the framework of the Alexander von Humboldt Professorship of TM Knight endowed by the German Federal Ministry of Education and Research.

Handling Editor: Giovanni Strona

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, analysing 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

*Tiffany M. Knight and Roberto Salguero-Gómez contributed equally to this work

This is an open access article under the terms of the Creative Commons Attribution-NonCommercial License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited and is not used for commercial purposes.

© 2021 The Authors. *Methods in Ecology and Evolution* published by John Wiley & Sons Ltd on behalf of British Ecological Society

1 | 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 (ESM, Table S1; Figure S1). 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 (a) the ability to model populations structured by continuously distributed traits, (b) 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), (c) efficient parameterization of demographic processes with familiar regression methods (Coulson, 2012), and (d) 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; Ellner et al., 2016; Merow et al., 2014) 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, 2020), 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 (Figure 1; Box 1; Tables 1 and 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 modelling trait-demography relationships, and enables users to specify IPMs of any functional form that they desire.

2 | 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

$$n(z', t + 1) = \int_L^U K(z', z) n(z, t) dz. \quad (1)$$

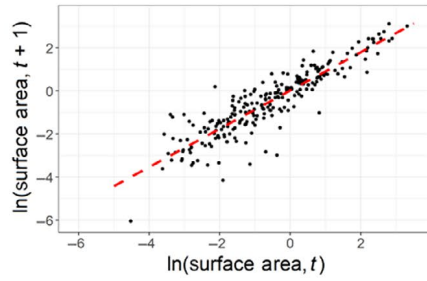
$K(z', z)$, known as the *projection kernel*, describes all possible transitions of existing individuals and recruitment of new individuals from t to $t + 1$, generating a new trait distribution $n(z', t + 1)$. L, U are the lower and upper bounds for values that the trait z can have, which defines the *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* (Equation 2). For example, a projection kernel to describe a life cycle where individuals can survive, transition to different state values, and reproduce via sexual and asexual pathways, can be split as follows.

$$K(z', z) = P(z', z) + F(z', z) + C(z', z), \quad (2)$$

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 (Equation 3) may contain two regression models: (a) a logistic regression of survival on log body mass (Equation 4) and (b)

Mathematical/graphical notation



Deterministic:

$$n(z', t+1) = \int_L^U K(z', z) n(z, t) dz$$

Parameter re-sampled stochastic:

$$n(z', t+1) = \int_L^U K(z', z, \theta) n(z, t) dz$$

Sub-kernel formula:

$$P(z', z) = s(z) * G(z', z)$$

Vital rate expressions:

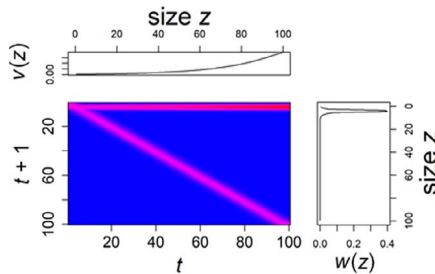
$$\text{Logit}(s) = \beta_0 + \beta_1 * z$$

$$G(z', z) = f_G(z', \mu(z), \sigma)$$

$$\mu(z) = \beta_0 + \beta_1 * z$$

$$L = 1.2, U = 7.8, N_{\text{mesh}} = 100$$

$$\theta \sim \text{Norm}(0, \sigma_{\text{Temp}})$$



ipmr/R representation

Fit vital rate models (1)

Other packages

(e.g. lme4, brms, mgcv, stats, nlme)

`lm(size_2 ~ size_1)`

Decide if IPM is (2)

simple/general,
density-(in)dependent,
deterministic/stochastic,
parameter/kernel stochastic`init_ipm("simple", "di", "det")``init_ipm("simple", "di",
"stoch", "param")`

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)`Numerically define (4)
kernels and initial conditions`define_ipml(),
define_domains(),
define_pop_state(),
define_env_state()`

Generate model object (5)

`make_ipm()`

Population level inference (6)

`lambda(), right_ev(), left_ev(),
mean_kernel(), plot(),
other packages (e.g. popbio,
popdemo)`

FIGURE 1 There are generally six 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 modelling. 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 analysed 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

a linear regression of log body mass at $t + 1$ on log body mass at t (Equations 5–6). 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.

$$P(z', z) = s(z) * G(z', z), \quad (3)$$

$$\text{Logit}(s(z)) = \alpha_s + \beta_s * z, \quad (4)$$

$$G(z', z) = f_G(z', \mu_G, \sigma_G), \quad (5)$$

BOX 1 Code to implement a simple IPM from parameter estimates in `ipmr`. Because `ipmr` does not include functions to assist with regression modelling, 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`.

```
library(ipmr)

# This section produces the result of Step 1 in Figure 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. 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. 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",
  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"))
)
```

BOX 1 (Continued)

```

# Step 4 in Figure 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 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.

example_ipm <- make_ipm(example_proto_ipm)

# Step 6 in Figure 1.

lambda(example_ipm)

```

TABLE 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)
$\text{logit}(s(z)) = \alpha_s + \beta_s * z$	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)
$\text{logit}(r_p(z)) = \alpha_{r_p} + \beta_{r_p} * z$	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)
$p_r = \frac{\# \text{Recruits}(t+1)}{\# \text{flowers}(t)}$	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 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 in bold 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
$\text{Logit}(s(z, a)) = \alpha_s + \beta_{s,z} * z + \beta_{s,a} * a$	<code>surv ~ size_1 + age, family = binomial()</code>	<code>s_age = plogis(s_int + s_z * z_1 + s_a * age)</code>
$G(z', z, a) = f_G(z', \mu_G(z, a), \sigma_G)$	<code>G = dnorm(size_2, mu_G_age, sigma_G)</code>	<code>G_age = dnorm(z_2, mu_G_age, sigma_G)</code>
$\mu_G(z, a) = \alpha_G + \beta_{G,z} * z + \beta_{G,a} * a$	<code>size_2 ~ size_1 + age, family = gaussian()</code>	<code>mu_G_age = G_int + G_z * z + G_a * age</code>
$\text{Logit}(m_p(z, a)) = \alpha_{m_p} + \beta_{m_p,z} * z + \beta_{m_p,a} * a$	<code>repr ~ size_1 + age, family = binomial()</code>	<code>m_p_age = plogis(m_p_int + m_p_z * z + m_p_a * age)</code>
$\text{Logit}(r_p(a)) = \alpha_{r_p} + \beta_{r_p,a} * a$	<code>recr ~ age, family = binomial()</code>	<code>r_p_age = plogis(r_p_int + r_p_a * age)</code>
$B(z', z) = f_B(z', \mu_B(z), \sigma_B)$	<code>b = dnorm(size_2, mu_rc_size, sigma_rc_size)</code>	<code>rc_size = dnorm(z_2, mu_rc_size, sigma_rc_size)</code>
$\mu_B(z) = \alpha_B + \beta_{B,z} * z$	<code>rc_size_2 ~ size_1, family = gaussian()</code>	<code>mu_rc_size = rc_size_int + rc_size_z * z</code>
$P_a(z', z) = s(z, a) * G(z', z, a)$		<code>P_age = s_age * g_age * d_z</code>
$F_a(z', z) = s(z, a) * m_p(z, a) * r_p(a) * B(z', z) / 2$		<code>F_age = s_age * f_p_age * r_p_age * rc_size / 2</code>
$n_0(z', t + 1) = \sum_{a=0}^{M+1} \int_L^U F_a(z', z) n_a(z, t) dz$		
$n_a(z', t + 1) = \int_L^U P_{a-1}(z', z) n_{a-1}(z, t) dz$		
$n_{M+1}(z', t + 1) = \int_L^U [P_{M+1}(z', z) n_{M+1}(z, t) + P_M(z', z) n_M(z, t)] dz$		

$$\mu_G = \alpha_G + \beta_G * z. \quad (6)$$

Analytical solutions to the integral in Equation 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 artificial size bins centered at z_i with width $h = (U - L) / m$. The midpoints $z_i = L + (i - 0.5) * h$ for $i = 1, 2, \dots, m$. The midpoint rule approximation for Equation 1 then becomes:

$$n(z_j, t + 1) = h \sum_{i=1}^m K(z_j, z_i) n(z_i, t). \quad (7)$$

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 and 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 and 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) modelled 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 et al. (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).

2.1 | A brief worked example of a simple IPM

Box 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 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 Equation 3, and the vital rates therein by Equations 4–6. The $F(z', z)$ kernel is given Equation 8:

$$F(z', z) = r_d(z') * r_n(z), \quad (8)$$

$$r_d(z') = f_{r_d}(z', \mu_{r_d}, \sigma_{r_d}), \quad (9)$$

$$\text{Log}(r_n(z)) = \alpha_{r_n} + \beta_{r_n} * z. \quad (10)$$

Equation 9 is a recruit size distribution (where f_{r_d} denotes a normal probability density function), and Equation 10 describes the number of new recruits produced by plants as a function of size z .

The code in Box 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.

2.2 | 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 (Baxter et al., 2006; Caswell, 2001; de Kroon et al., 1986; 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 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.

2.3 | 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 2). The suffixes appended to each variable name in the *ipmr* formulation correspond to the subscript- and/or

superscript 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.

3 | 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 modelling 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. We are in the process of creating the PADRINO IPM database using *ipmr* and `proto_ipms` 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., 2016; Salguero-Gómez et al., 2014). Recent work has highlighted the power of syntheses that harness many structured population models (Adler et al., 2014; Compagnoni et al., 2021; Salguero-Gómez et al., 2016). 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.

ACKNOWLEDGEMENTS

We thank the Associate Editor and two anonymous reviewers for comments that greatly improved this manuscript.

CONFLICTS OF INTEREST

The authors declare no conflicts of interest.

AUTHORS' CONTRIBUTIONS

All authors contributed to package design. S.C.L. implemented the package. All authors wrote the first draft of the manuscript and contributed to revisions.

PEER REVIEW

The peer review history for this article is available at <https://publons.com/publon/10.1111/2041-210X.13683>.

DATA AVAILABILITY STATEMENT

The *Carpobrotus* dataset is included in the *ipmr* R package. The package is available on GitHub at <https://github.com/levisc8/ipmr>, CRAN at <https://cran.r-project.org/web/packages/ipmr/index.html> (Levin et al., 2021), and Zenodo at <https://doi.org/10.5281/zenodo.5095062> (Levin, 2021). The paper and case studies do not use any other data.

ORCID

Sam C. Levin  <https://orcid.org/0000-0002-3289-9925>
 Dylan Z. Childs  <https://orcid.org/0000-0002-0675-4933>
 Aldo Compagnoni  <https://orcid.org/0000-0001-8302-7492>
 Sanne Evers  <https://orcid.org/0000-0002-8002-1658>
 Tiffany M. Knight  <https://orcid.org/0000-0003-0318-1567>
 Roberto Salguero-Gómez  <https://orcid.org/0000-0002-6085-4433>

REFERENCES

- Adler, P. B., Ellner, S. P., & Levine, J. M. (2010). Coexistence of perennial plants: An embarrassment of niches. *Ecology Letters*, 13, 1019–1029. <https://doi.org/10.1111/j.1461-0248.2010.01496.x>
- 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 of the United States of America*, 111(2), 740–745. <https://doi.org/10.1073/pnas.1315179111>
- Baxter, P. W. J., McCarthy, M. A., Possingham, H. P., Menkhurst, 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>
- 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>
- Bruno, J. F., Ellner, S. P., Vu, I., Kim, K., & Harvell, C. D. (2011). Impacts of aspergilliosis on sea fan coral demography: Modeling a moving target. *Ecological Monographs*, 81(1), 123–139. <https://doi.org/10.1890/09-1178.1>
- Caswell, H. (2001). *Matrix population models: Construction, analysis, and interpretation* (2nd ed.). Sinauer Associates Inc.
- 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>
- 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: Biological Sciences*, 271(1547), 425–434. <https://doi.org/10.1098/rpsb.2003.2597>
- 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>
- 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>
- 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>
- 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>
- de Kroon, H., Plaisier, A., van Goenendaal, J., & Caswell, H. (1986). Elasticity: The relative contribution of demographic parameters to population growth rate. *Ecology*, 67(5), 1427–1431.
- Easterling, M. R., Ellner, S. P., & Dixon, P. M. (2000). Size specific sensitivity: Applying a new structured population model. *Ecology*, 81(3), 694–708.
- Ellner, S. P., Childs, D. Z., & Rees, M. (2016). *Data-driven modelling of structured populations: A practical guide to the integral projection model*. Springer International Publishing AG.
- Ellner, S. P., & Rees, M. (2006). Integral projection models for species with complex demography. *The American Naturalist*, 167(3), 410–428.
- 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>
- 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. <https://doi.org/10.1007/s10144-012-0308-7>
- 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>
- 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>
- Levin, S. C. (2021). Data from: *Levis8/ipmr*: (Version v0.0.3). Zenodo, <https://doi.org/10.5281/zenodo.5095062>
- Levin, S. C., Compagnoni, A. C., Childs, D. Z., Evers, S., Salguero-Gomez, R., & Knight, T. M. (2021). *ipmr: Fits Integral projection models using an expression based framework*. R package version 0.0.2. <https://CRAN.R-project.org/package=ipmr>
- 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.121465>
- 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>
- 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>
- 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., ... Vaupel, J. W. (2016). COMADRE: A global database of animal demography. *Journal of Animal Ecology*, 85, 371–384. <https://doi.org/10.1111/1365-2656.12482>
- Rees, M., & Ellner, S. P. (2009). Integral projection models for populations in temporally varying environments. *Ecological Monographs*, 79(4), 575–594. <https://doi.org/10.1890/08-1474.1>.
- 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., ... Vaupel, J. W. (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>
- 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>
- 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>

SUPPORTING INFORMATION

Additional supporting information may be found online in the Supporting Information section.

How to cite this article: Levin, S. C., Childs, D. Z., Compagnoni, A., Evers, S., Knight, T. M., & Salguero-Gómez, R. (2021). ipmr: Flexible implementation of Integral Projection Models in R. *Methods in Ecology and Evolution*, 00, 1–9. <https://doi.org/10.1111/2041-210X.13683>

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

Sam C. Levin ^{*1,2,3}, Sanne Evers ^{1,2}, Tomos Potter³, Mayra Peña Guerrero ^{1,2}, Dylan Z. Childs ⁴, Aldo Compagnoni ^{1,2,3}, Tiffany M. Knight ^{1,2,5†}, Roberto Salguero-Gómez ^{3‡}

¹Institute of Biology, Martin Luther University Halle-Wittenberg, Am Kirchtor 1, 06108 Halle (Saale), Germany

²German Centre for Integrative Biodiversity Research (iDiv) Halle-Jena-Leipzig, Deutscher Platz 5e, 04103 Leipzig, Germany

³Department of Zoology, 11a Mansfield Rd, University of Oxford, Oxford, OX1 3SZ, UK

⁴Department of Animal and Plant Sciences, University of Sheffield, Sheffield, S10 2TN, UK

⁵Department of Community Ecology, Helmholtz Centre for Environmental Research-UFZ, Theodor-Lieser-Straße 4, 06120, Halle (Saale), Germany

† Joint senior authors

Abstract

1. Discrete time structured population projection models are an important tool for studying population dynamics. Within this field, Integral Projection Models (IPMs) have become a popular method for studying populations structured by continuously distributed traits (*e.g.* height, weight). Databases of discrete time, discrete state structured population models, for example DATLife (life tables) and COMPADRE & COMADRE (matrix population models), have made quantitative syntheses straightforward to implement. These efforts allow researchers to address questions in both basic and applied ecology and evolutionary biology. There are now over 300 peer-reviewed publications containing IPMs. I describe a novel method to quickly reconstruct these models for subsequent analyses with PADRINO, a new demographic database.
2. I introduce the open access database of IPMs, PADRINO, and its accompanying R package, *Rpadrino*, which enables users to download, subset, reconstruct, and extend published IPMs. *Rpadrino* makes use of recently created software, *ipmr*, to provide an engine to reconstruct a wide array of IPMs from their symbolic representations and conduct subsequent analyses.
3. In the first release (v0.0.1), PADRINO houses 280 IPMs from 40 publications that describe the demography of 14 animal and 26 plant species. All IPMs in PADRINO are tested to ensure they reproduce published estimates. Furthermore, PADRINO is designed to allow users to add data from external sources and modify parameter values, providing a platform for extending models beyond their original purpose while allowing for full reproducibility.
4. PADRINO and *Rpadrino* provide a toolbox for asking new questions and conducting syntheses with peer-reviewed published IPMs. *Rpadrino* provides a user-friendly interface so researchers do not need to worry about the database structure or syntax, and can focus on their research questions and analyses. Additionally, *Rpadrino* is thoroughly documented, and provides numerous examples of how to perform analyses which are not included in the package's functionality.

Keywords: Elasticity, database, demography, open access, population dynamics, life history, sensitivity

Introduction

Demography provides an excellent approach to examine the ecology (Crone et al. 2011), evolutionary biology (Metcalf & Pavard 2007), and conservation biology of any species (Doak & Morris 2001). Environmental conditions and biotic interactions influence vital rates (*e.g.* survival, development, reproduction) across the

entire life cycle, which then govern its short-term and long-term performance (Caswell 2001). A variety of methods exist for summarizing vital rates into demographic models; discrete-time, structured population models are among the most popular (Crone et al. 2011, Caswell 2001). Indeed, there is a rich history of using such structured population models across a variety of sub-disciplines in ecology (*e.g.* Leslie 1945, Caswell 2001, Easterling et al. 2000, Adler et al. 2010, Ellner et al. 2016).

Matrix projection models (MPMs) are the most widely used structured population model. MPMs divide the population into discrete classes corresponding to some trait value (*e.g.* developmental state, age, or size), and then model the population using vital rates computed for each class. The popularity of MPMs and the desire to conduct quantitative syntheses led to the creation of databases that enable researchers to reproduce and analyze published MPMs en masse (Salguero-Gómez et al. 2014, Salguero-Gómez et al. 2015).

Recent advances in the availability of this structured demographic information have powered large scale syntheses to answer a variety of questions in both theoretical and applied ecology. For example, Jelbert and colleagues (2019) used the COMPADRE database (Salguero-Gómez et al. 2014) to show that the intrinsic ability of plant populations to benefit from disturbance in the short term (amplification *sensu* Stott et al. 2011) predicts invasiveness in novel environments. Similarly, Healy and colleagues (2019) used the COMADRE database animal MPMs (Salguero-Gómez et al. 2015) to demonstrate how patterns of animal life history strategies are shaped by trade-offs between distributions of age-specific mortality and reproduction. Using DATLife (DATLife Database 2021), an open access repository of life tables, Hartemink & Caswell (2018) decomposed the variance in individual longevity into contributions from individual heterogeneity and individual stochasticity in 10 invertebrate species.

For many species, vital rates are best predicted as a function of one or more continuous trait (*e.g.* size, height, mass), rather than as a function of discrete classes. Integral projection models (IPMs), which are continuously structured population models, have become an increasingly important tool for ecologists interested in addressing broad biological questions through a demographic lens (Gonzalez et al. 2021). IPMs combine vital rate functions of continuous traits into projection kernels, which describe how the abundance and distribution of trait values in a population change in discrete time (Easterling et al. 2000). IPMs have been used to investigate a variety of topics, such as invasive species spread (*e.g.* Jongejans et al. 2011, Erickson et al. 2017), evolutionary stable strategies (*e.g.* Childs et al. 2004), the effect of climate drivers on population persistence (Salguero-Gómez et al. 2012, Compagnoni et al. 2021a), and linking evolutionary feedbacks to population dynamics (Coulson et al. 2011). To date, there are at least 300 publications covering over 380 species (Figure 2.1). Given the existing volume of data and its rate of increase, the field of ecology would benefit from a centralized repository that enables researchers to reconstruct and analyze these models to address a multitude of synthetic questions.

Here, I introduce PADRINO and *Rpadrino*. PADRINO is an open access database of IPMs. Specifically, PADRINO houses symbolic representations of IPMs, their parameter values, and associated metadata to aid users in selecting appropriate models. *Rpadrino* is an R package that enable users to reconstruct and modify IPMs from PADRINO. In the following, I describe PADRINO’s structure and how to efficiently interact with it using *Rpadrino*. Next, I explain how IPMs are digitized into the database, along with metadata and associated assumptions and challenges. I conclude with discussion of future directions for the database. Importantly, our Supplementary Materials contain two case studies. The first case study demonstrates how to use PADRINO and *Rpadrino* to reconstruct published IPMs, conduct perturbation analyses, compute some life cycle events, and troubleshoot problems. The second case study shows how to use PADRINO, *Rpadrino*, and *ipmr* to combine PADRINO IPMs with user-specified IPMs, and then how to use PADRINO data with other databases using BIEN (Maitner et al. 2017) and COMPADRE as examples, demonstrating the potential for PADRINO in broad, macro-ecological applications.

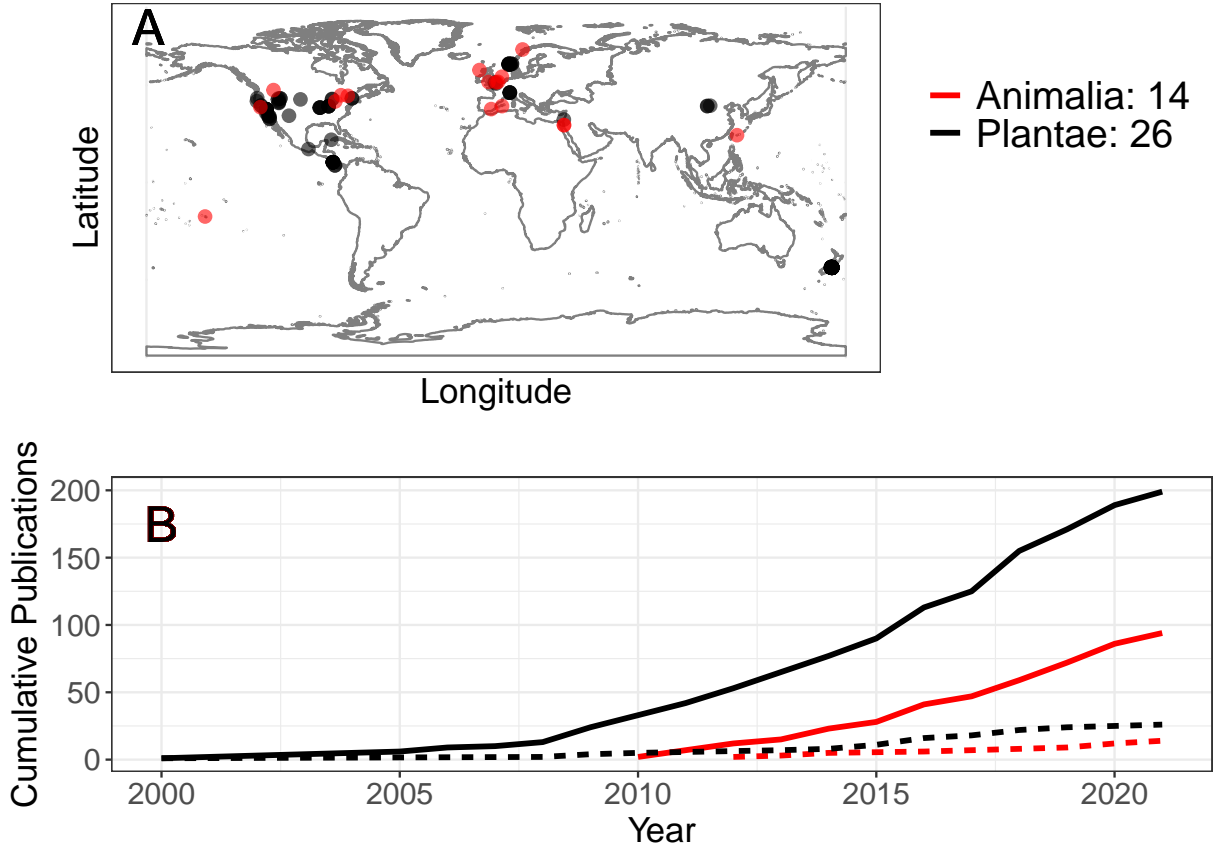


Figure 2.1: The geographic and temporal coverage of studies in the PADRINO IPM Database. (A) Geographic distribution of publications currently contained in PADRINO (i.e. studies from Table 2). (B) Cumulative number of publications found by our search criteria by year (solid lines), and the number that are in the released version of PADRINO (dashed lines). Future releases will include those that I have found, but are not yet completely digitized (i.e. those represented by solid lines, but not yet included in the dashed lines). See the Supplementary Data for a complete list of IPM publications.

PADRINO

Chapter 1 provides an introduction to how IPMs are created, and so I will not review that here. Existing approaches to digitizing structured population models entail entering individual transition and reproductive contributions of discrete stages, which are represented by scalar values (*e.g.* COMPADRE and COMADRE). Entering the scalar transition probabilities and reproductive contributions would be difficult for IPMs whose kernels are approximated with relatively small iteration matrices (*e.g.* 45×45), and impossible for more typical sizes (100×100 - 1000×1000). Furthermore, there is a greater range of subsequent analyses which are possible by capturing the underlying model, rather than its numerical approximation. The recent development of more advanced computational tools, namely *rlang* (Henry & Wickham 2021) and *ipmr* (Levin et al. 2021), mean we can now create IPMs from their symbolic representations and the parameters used to implement them. A database containing the functional forms of sub-kernels and vital rate functions, and the parameter values that go into each of those, could accommodate *nearly* all of the existing IPM literature.

PADRINO is an open-access database of integral projection models. PADRINO defines a syntax to symbolically represent IPMs as text strings, and stores the values of those symbols in a separate table. The syntax used is

very similar to the mathematical notation of IPMs, and is largely “language-agnostic” (*i.e.* aims to avoid idiosyncrasies of specific programming languages). For example, a survival/growth kernel with the form $P(z', z) = s(z) * G(z', z)$ would be `P = s * G` in PADRINO’s syntax. $G(z', z) = f_G(z' | \mu_g(z), \sigma_G)$ (where f_G denotes a normal probability density function) becomes `G = Norm(mu_g, sd_g)`. This notation should be translatable to many computing languages beyond just *R* (*e.g.* Python or Julia).

PADRINO v0.0.1 consists of 10 tables (Table 1, Figure S1). In this first version, PADRINO currently contains 280 IPMs from 40 peer-reviewed publications that consider 14 animal and 26 species (Table 2). However, I highlight that PADRINO is under active development, and I continue to digitize studies for release in future versions (see Supplementary Data). These tables form a database, with tables linked using a common column across all tables: `ipm_id`. The scope of each `ipm_id` is determined by the way that an IPM is parameterized. IPMs that characterize the same species across, for example, many years or sites, with the same functional form, are included under a single `ipm_id`. For instance, a growth model that includes a random intercept for different years could be used to generate many unique projection kernels. These are stored under a single `ipm_id` because the functional form of the IPM is identical for each year, and only the parameter values change. One exception to this grouping rule is when the sites (*i.e.* where the raw data are reported to have come from) are far enough apart that separate sets of GPS coordinates are used to describe them. These IPMs are split into separate `ipm_ids` so that the spatial distinctions are preserved, which facilitates matching PADRINO data with, for example, gridded environmental data (*e.g.* Compagnoni et al. 2021b, Case Study 2).

Finally, there are two important details potential users should be aware of. The first detail is that PADRINO provides IPMs *as they are published following peer review*. I do not alter these IPMs when digitizing them, except to correct typographical errors that may have found their way into the peer-reviewed publication. The second detail is that PADRINO does not store any raw data used to create the IPMs. Users should be aware of these, and I encourage all users to consult and cite the original publications of each IPM before including it in an analysis.

The Digitization Process

The IPM digitization process begins when a peer-reviewed paper containing an IPM is published. I have set alerts for the following keyword searches: “Integral Projection Model OR IPM OR sensitivit* OR elasticit* OR Vital rate OR LTRE”. This automatic weekly search is run on Google Scholar and Scopus, and resulting hits are examined manually to find publications that contain an IPM. Once a paper containing an IPM is identified, I extract five types of metadata: taxonomic information (*e.g.* species names, functional groups), publication information (*e.g.* authors, complete citation, year of publication), temporal metadata (*e.g.* study duration, data collection beginning and ending months and years), spatial metadata (*e.g.* latitude/longitude, ecoregion), and model specific metadata (*e.g.* experimental treatments applied, density-(in)dependent). Table 3 contains a complete description of the metadata table in PADRINO.

Following the metadata digitization, I extract functional forms of each sub-kernel, vital rate function, and how the environment varies (if applicable). The functional forms of each component of the model are expressed in the syntax introduced above. Finally, I extract all of the parameter values, as well as information on the range of values each trait can take on and how they are numerically approximated (*i.e.* integration rules). The parameter values and integration information are then substituted for symbol names when the user requests a built model. For example, in *Rpadrino*, the `Norm(mu_g, sd_g)` from above would be translated to `dnorm(z_2, mu_g, sd_g)` (see *Rpadrino* below for more details).

Often times, not all of the required information is present in the publication or its supplementary materials. Therefore, I often contact authors to request the required information and/or ask for clarification. I also extract a target value for the data validation step (see next section), so that I can ensure that released data really does replicate the published IPM. A complete guide to our digitization process and documentation of the database syntax is publicly available on PADRINO’s webpage (<https://padrinoDB.github.io/Padrino/>).

Data Validation and Reproducibility

The PADRINO IPM Database has automated testing built into the data release process. All IPMs are checked to ensure they recover the behavior of the published version prior to release. In most cases, validation consists of reproducing the kernel-specific asymptotic population growth rate (λ) to within ± 0.03 of the published λ value in the source publication. It is worth noting that this margin of error is considerably lower than the uncertainty that arises from fitting statistical models to the raw data used in the IPM (*e.g.* Clark 2003), and so it should be acceptable for almost any application. For stochastic models with continuously varying environments, it is often not computationally feasible to re-run the IPM for 10-50,000 iterations since they are time consuming to run and there are many in PADRINO. Thus, I manually check for shorter term behavior that is similar to published dynamics (*e.g.* stochastic population growth rate (λ_s) after 1000 iterations). For publications where population growth rates are not available, I manually examine the publication and check the model digitized in PADRINO against some reported behavior (*e.g.* generation time). A given IPM can only enter a scheduled database release if it is explicitly flagged by a digitizer as validated, or if it passes its automated test. The manual testing functionality is contained in the open source *R* package *pdbDigitUtils* (available on GitHub (<https://github.com/padrinoDB/pdbDigitUtils>)), and PADRINO's build scripts are in the project's GitHub repository (<https://github.com/padrinoDB/Padrino/tree/main/R>).

Rpadrino

Rpadrino is an *R* package that contains functions for downloading the PADRINO database, data querying and management, and modifying existing functional forms and parameter values. Additionally, *Rpadrino* provides wrappers around functions from *ipmr* (Levin et al. 2021), which is a general engine for creating IPMs from functional forms and parameter estimates in *R*. These wrappers translate PADRINO's syntax into *ipmr*'s syntax, enabling users to select appropriate models based on the provided metadata, and then reconstruct the IPMs with only a few function calls. *Rpadrino* uses *ipmr*'s `proto_ipm` data structure as an intermediate step between a set of functional forms and parameters, and an implemented IPM in both *ipmr* and *Rpadrino*. The intermediate step enables users to combine their own IPMs with ones from PADRINO. Finally, *Rpadrino* provides methods for PADRINO objects for the generic analysis functions in *ipmr*. Figure 2.2 provides an overview of the relationships between each of these packages.

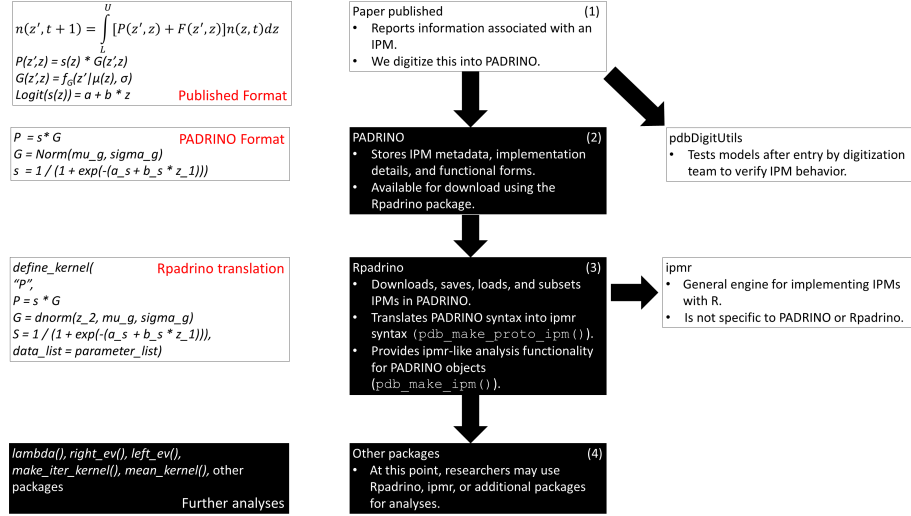


Figure 2.2: An overview of the relationships between the *PADRINO* IPM Database, and the *Rpadrino*, and *ipmr* R packages. *PADRINO* stores a general syntax for representing IPMs, which *Rpadrino* translates into *R* code that makes use of *ipmr*. The *pdbDigitUtils* package provides utilities to the digitizing team to ensure that models are entered correctly. Arrows show the flow of information through the process of digitizing, downloading, and analyzing data. Black boxes denote user-facing steps. Briefly, a model is digitized and then checked following publication (1). *Rpadrino* downloads the *PADRINO* IPM database, and translates *PADRINO* IPMs into usable *R* code (2). The *ipmr* R package is then used to generate actual IPM objects. Users do not necessarily need to know how to use *ipmr* to use *PADRINO* because *Rpadrino* provides wrapper functions around most of *ipmr*’s functionality (3). Once an IPM or set of IPM objects are created, users can call functionality from *Rpadrino*, *ipmr*, or other packages to address their broader research questions (4).

The flexibility of IPMs and their broad application across ecology, evolution, and conservation biology mean that there is no fixed set of steps in a workflow using *Rpadrino*. However, there are generally four steps that a researcher must take when using these tools. The first step is to identify studies of interest, and, optionally, augment *PADRINO*’s metadata with additional information from other sources (*e.g.* environmental data, GBIF). *Rpadrino*’s representation of *PADRINO* objects is designed to accommodate the range of further analyses that researchers may be interested in. Users may augment any table with additional information corresponding to, for example, spatial or temporal covariates from other open access databases.

The second step in an *Rpadrino* workflow is to construct a list of `proto_ipm` objects using `pdb_make_proto_ipm()` (Figure 2.2). This function translates *PADRINO*’s syntax into *ipmr* code, and then builds a `proto_ipm` object for each unique `ipm_id`. For some models, users may choose to create deterministic or stochastic IPMs at this step. *Rpadrino*’s default behavior is to generate deterministic models whenever possible. This behavior encompasses instances where authors generated models with no time or space varying parameters, and where authors included discretely varying environments. The latter can be implemented as deterministic models because all parameter values are known before the model is built. Models with continuous environmental variation require sampling the environment at each model iteration, usually by sampling from distributions randomly. These are always considered stochastic models.

The third step in an *Rpadrino* workflow is creating IPM objects with `pdb_make_ipm()` (Figure 2.2). `pdb_make_ipm()` uses *ipmr*’s `make_ipm()` function to build IPM objects. Users may specify additional options to pass to `make_ipm()` (*e.g.* normalize the population size to always equal 1, return the vital rate function values as well as the sub-kernels and population state). The various arguments users can modify are described in the *ipmr* documentation for `make_ipm()`.

The fourth and final step in an *Rpadrino* workflow is to conduct the analyses of interest. *Rpadrino* provides wrappers around *ipmr* functions to extract per-capita growth rates, eigenvectors (Caswell 2001, Ellner, Childs

& Rees 2016 Ch. 2), assess convergence to asymptotic dynamics (Caswell 2001), compute mean kernels for stochastic IPMs (Ellner, Childs & Rees Ch 7), and modify existing IPMs with new parameter values and functional forms. Additionally, the documentation on the [Rpadrino website](#) and the Supplementary Materials for this paper contain details on how to conduct more complicated analyses with IPM objects (*e.g.* perturbation analyse (Ellner, Childs & Rees 2016 Ch 4), size at death calculations (Metcalf et al. 2009)). The package documentation and the recent publication describing *ipmr* also contain code demonstrating analyses on single IPM objects (Levin et al. 2021). These can be extended via the **apply** family of functions.

Challenges

Digitizing IPMs into the PADRINO IPM Database is not without issues. First, it is often the case that the complete form of the IPM is not reported: approximately 80% of papers I have examined thus far fall into this category. Many studies may report the general form of the model (*e.g.* $n(z', t + 1) = \int_L^U K(z', z)n(z, t)dz$), but do not then report the functional forms of the sub-kernels or vital rates. Without the functional form of all vital rates and sub-kernels, it is impossible to reproduce the IPM. Second, some parameter values may be missing from the main text or supplementary materials - common culprits are terms for the variation of the growth/fecundity kernels, number of meshpoints, and integration bounds (i.e. L, U in Eq 1). The authors of this paper have been guilty of this, as well as other sins of omission, in their own IPM publications. The intent here is not to alienate other authors, but offer a gentle reminder that reporting all parameter values and functional forms can go a long way towards making their science reusable and extensible. Reproducible science can often bring great benefit to the original authors as well as the broader community (Kousta et al. 2019).

In addition to the challenges on the data digitization side, important challenges remain in the reconstruction of IPMs. Semi- or non-parametric models may be used to generate IPMs, and I have not yet defined a syntax in PADRINO for representing these vital rate models, though work is ongoing. Additionally, *ipmr* is not yet able to handle two-sex models (*e.g.* Stubberud et al. 2019), time-lagged models (*e.g.* Rose et al. 2005), or periodic models (*e.g.* Letcher et al. 2014). These types of IPMs do not yet represent a substantial portion of the literature. Nonetheless, it is our intention to continue developing functionality to accommodate them in future releases of PADRINO, *ipmr*, and *Rpadrino*.

Opportunities and Future Directions

PADRINO is designed to be compatible with other databases. Its metadata table closely resembles that of COMPADRE and COMADRE (Salguero-Gómez et al. 2015). This means that users who are familiar with these matrix population model databases should find a smooth transition to working with PADRINO. Furthermore, I provide a comprehensive set of metadata that allows users to match information in PADRINO with other databases (*e.g.* gridded climate data, functional trait information, phylogenetic information). These features will enable researchers to carry out more detailed and comprehensive analyses at various spatial, temporal, and phylogenetic scales.

PADRINO presents unique opportunities for synthesis in both theoretical and applied contexts. The expanded range of phylogenetic and geographical coverage can be used in conjunction with other demographic databases (*e.g.* COM(P)ADRE (Salguero-Gómez et al. 2015, Salguero-Gómez et al. 2016), popler (Compagnoni et al. 2019), DatLife (DatLife 2021)) to power larger scale syntheses than were possible before (*e.g.* Compagnoni et al. 2021b). For example, one could use IPMs from PADRINO and matrix population models from COMPADRE and COMADRE to create life tables (Jones et al. 2021), which could then be combined with life tables from DatLife for further analysis (*e.g.* Jones et al. 2014). The intermediate life table conversion steps may not be necessary, as many of the same life history traits and population level parameters may be calculated from all of these models (Caswell 2001, Ellner, Childs & Rees 2016). Furthermore, recent publications combine biotic and abiotic interactions into demographic models providing a robust theoretical toolbox for exploring species responses to environmental drivers such as climate change (*e.g.* Simmonds et al. 2020, Abrego et al. 2021). *Rpadrino* also provides functionality to modify parameter values and functional

forms of the IPMs it stores, giving theoreticians a wide array of realistic life histories to experiment with. These examples are far from an exhaustive list, but hopefully demonstrates the potential for this new tool in demography, ecology, and evolutionary biology.

Citations

1. Easterling, M.R., Ellner, S.P., & Dixon, P.M. (2000). Size specific sensitivity: applying a new structured population model. *Ecology* 81(3): 694-708.
2. Ellner, S.P. & Rees, M. (2006). Integral Projection Models for species with complex demography. *The American Naturalist* 167(3): 410-428.
3. Hartemink, N., & Caswell, H. (2018). Variance in animal longevity: contributions of heterogeneity and stochasticity. *Population Ecology* 60: 89-99. <https://doi.org/10.1007/s10144-018-0616-7>
4. DATLife – The Demography Across the Tree of Life – database. Max-Planck Institute for Demographic Research (Germany). Available at www.datlife.org. (2021)
5. 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., Viereg, 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>
6. 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>
7. Compagnoni, A., Pardini, E., & Knight, T.M. (2021a). Increasing temperature threatens an already endangered coastal plant species. *Ecosphere* 12(3): e03454. <https://doi.org/10.1002/ecs2.3454>
8. 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-Gómez, R., & Knight, T.M. (2021b). 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>
9. Levin, S.C., Childs, D.Z., Compagnoni, A., Evers, S., Knight, T.M., & Salguero-Gómez, R. (2021) ipmr: Flexible implementation of Integral Projections Models in R. *Methods in Ecology and Evolution*. <https://doi.org/10.1111/2041-210X.13683>
10. 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>
11. Jelbert, K., Buss, D., McDonald, J., Townley, S., Franco, M., Stott, I., Jones, O., Salguero-Gómez, R., Buckley, Y., Knight, T.M., Silk, M., Sargent, F., Rolph, S., Wilson, P., & Hodgson, D. (2019). Demographic amplification is a predictor of invasiveness among plants. *Nature Communications* 10: 5602. <https://doi.org/10.1038/s41467-019-13556-w>
12. 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>
13. 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>
14. 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
15. Leslie, P.H. (1945). The use of matrices in certain population mathematics. *Biometrika* 33(3): 183-212.

****Table 2.1**:** Summaries of the information contained in each table of the PADRINO database. A complete guide to each column in each table is available on the project's webpage in the form of the guide provided to digitizers (there are too many columns to provide the information here).

Table	Description
Metadata	This table contains metadata for each IPM. This is organized into taxonomic information (full taxonomy plus functional group information), publication information (citation, authorship, source), data collection information (study period/duration, GPS coordinates, ecoregion), and model specific information (studied sexes, eviction corrections, treatments applied, and model implementation details). See Table 3 for more information on these columns.
State Variables	This table contains the names of the state variables used in the model and whether or not they are discrete or continuously distributed.
Continuous States	This table contains names and ranges for each continuously distributed state variable in the model, as well as which kernels they apply to (kernels are the $P(z',z)$, $F(z',z)$, and $C(z',z)$ in Eq 1).
Integration Rules	This table contains information on how each continuous state variable is numerically approximated in the model (i.e. number of meshpoints, which integration rule was used).
Population Trait Distributions	This table contains the names of the population trait distributions used in the model ($n(z,t)$ and $n(z',t+1)$ in Eq. 1).
IPM Sub-kernels	This table contains the functional forms of each sub-kernel in the IPM (e.g. $P(z',z)$ in Eq 1 becomes ' $P = s * G$ '), and information on which traits it acts on and creates. This table makes use of [parameter set index notation](https://levisc8.github.io/ipmr/articles/index-notation.html) from 'ipmr' to concisely represent models which may produce many kernels.
Vital Rate Functions	This table contains the functional forms of each vital rate in the IPM (e.g. ' $\mu_g = \text{int}_g + \text{slope}_g * z_1$ '). This table makes use of [parameter set index notation](https://levisc8.github.io/ipmr/articles/index-notation.html) from 'ipmr' to concisely represent models which may produce many kernels.
Parameter Values	This table contains the names and values of each parameter in the model, with the exception of parameters that are associated with continuous environmental variation.
Continuous Environmental Variation	This table contains parameter values and functional forms of any continuously varying environmental conditions (e.g. yearly variation in precipitation and/or temperature). Any model that contains information in this table is considered stochastic by default, as these variables must be sampled at least once to construct a model with 'Rpadrino'.
Parameter Set Indices	This table contains the parameter set indices. These are substituted into the IPM kernels and vital rate expressions when a model is built, so that a single symbolic expression can represent an arbitrary number of realized expression. For example, the vital rate expression ' $\mu_g = \text{int}_g + \text{slope}_g * z_1$ ' can be used to represent a range of years for a model with year-specific intercepts. This table contains values substituted in for ' $_{yr}$ ' across the model. See the [ipmr vignette on Index Notation](https://levisc8.github.io/ipmr/articles/index-notation.html) for more details.

****Table 2.2**:** Taxonomic representation of IPMs in PADRINO. These numbers represent the number of models that are error checked and accurately reproduce the published IPM (see 'Data Validation' for more details). Models that are partially entered or still contain errors are not considered here. I am in the process of correcting them and/or retrieving additional information from the authors.

Kingdom	# of Unique ipm_id's	# of Unique Species	# of Publications
Totals	280	56	40
Animalia	22	16	14
Plantae	258	40	26

16. Caswell, H. (2001) Matrix population models: construction, analysis, and interpretation, 2nd edn. Sunderland, MA: Sinauer Associates Inc
17. Adler, P.B., Ellner, S.P. & Levine, J.M. (2010). Coexistence of perennial plants: an embarrassment of niches. *Ecology Letters* 13: 1019-1029. <https://doi.org/10.1111/j.1461-0248.2010.01496.x>
18. Abrego, N., Roslin, T., Huotari, T., Ji, Y., Schmidt, N.M., Wang, J., Yu, D.W., & Ovaskainen, O. (2021) Accounting for species interactions is necessary for predicting how arctic arthropod communities respond to climate change. *Ecography*. <https://doi.org/10.1111/ecog.05547>
19. Henry, L., & Wickham, H. (2021). rlang: Functions for Base Types and Core R and 'Tidyverse' Features. R package version 0.4.11. <https://CRAN.R-project.org/package=rlang>
20. Crone, E.E., Menges, E.S., Ellis, M.M., Bell, T., Bierzychudek, P., Ehrlén, J. et al. (2011) How do ecologists use matrix population models? *Ecology Letters* 14(1): 1-8. DOI: <https://doi.org/10.1111/j.1461-0248.2010.01540.x>
21. Doak, D. & Morris W.F. (2002). Quantitative Conservation Biology: Theory and Practice of Population Viability Analysis. Oxford, UK: Oxford University Press
22. RSG + Gamelon M. (2021). Demographic Methods Across the Tree of Life. Oxford, UK: Oxford University Press
23. Coulson, T., MacNulty, D.R., Stahler, D.R., von Holdt, B., Wayne, R.K., & Smith, D.W. (2011). Modeling effects of environmental change on wolf population dynamics, trait evolution, and life history. *Science* 334(6060): 1275-1278. <https://doi.org/10.1126/science.1209441>
24. Stott, I., et al. (2011). A framework for studying transient dynamics of population projection matrix models. *Ecology Letters* 14: 959-970. DOI: <https://doi.org/10.1111/j.1461-0248.2011.01659.x>
25. Clark, J.S. (2003) Uncertainty and variability in demography and population growth: A hierarchical approach. *Ecology* 84(6): 1370-1381.
26. Letcher, B.H., Schueller, P., Bassar, R.D., Nislow, K.H., Coombs, J.A., Sakrejda, K. *et al.* (2014). Robust estimates of environmental effects on population vital rates: an integrated capture-recapture model of seasonal brook trout growth, survival and movement in a stream network. *Journal of Animal Ecology* 84(2): 337-352. <https://doi.org/10.1111/1365-2656.12308>
27. Støttrup, M.W., Vindenes, Y., Vollestad, L.A., Winfield, I.J., Stenseth, N.C., & Langangen, O. (2019). Effects of size- and sex-selective harvesting: an integral projection model approach. *Ecology and Evolution* 9: 12556-12570. <https://doi.org/10.1002/ece3.5719>
28. Maitner, B., Boyle B., Casler N., Condit R., Donoghue J., Duran S.M., *et al.* (2017) The bien r package: A tool to access the Botanical Information and Ecology Network (BIEN) database. *Methods in Ecology and Evolution* 9(2): 373-379. <https://doi.org/10.1111/2041-210X.12861>
29. Simmonds E.G., Cole, E.F., Sheldon, B.C., & Coulson, T. (2020). Phenological asynchrony: a ticking time-bomb for seemingly stable populations? *Ecology Letters* 23(12): 1766-1775. <https://doi.org/10.1111/ele.13603>

****Table 2.3****: All columns contained in the Metadata table.

Concept	Column Name	Description
	ipm_id	Unique ID for each model.
Taxonomy	species_author	The Latin species name used by the authors of the paper.
	species_accepted	The Latin species name accepted by Catalogue of Life.
	tax_genus	The genus name accepted by Catalogue of Life.
	tax_family	The family name accepted by Catalogue of Life.
	tax_order	The order name accepted by Catalogue of Life.
	tax_class	The class name accepted by Catalogue of Life.
	tax_phylum	The phylum name accepted by Catalogue of Life.
	kingdom	The kingdom name accepted by Catalogue of Life.
	organism_type	General functional type of the species (_e.g._ annual, fern, mammal, reptile).
	dicot_monocot	If a plant species, whether the species is a dicot or a monocot.
	angio_gymno	If a plant species, whether the species is an angiosperm, gymnosperm, or neither.
Source	authors	All of a study authors' last names, separated by ';'
	journal	Abbreviated journal name (www.abbreviations.com/jas.php), or 'PhD', 'MSc' if a thesis.
	pub_year	The year of publication.
	doi	Digital object identifier and/or ISBN (if available).
	corresponding_author	The name of the corresponding author on the paper.
	email_year	The email address of the corresponding author and the year it was extracted (some email addresses may be defunct now).
	remark	Additional remarks from the digitizer regarding the publication, if any.
	apa_citation	The full APA citation for the source.
	demog_appendix_link	The URL for the Supplementary information containing additional model details, if available.
Temporal Metadata	duration	The duration of the study, defined 'study_end - study_start + 1'. Does not consider skipped years.
	start_year	The year demographic data collection began.
	start_month	The month demographic data collection began.
	end_year	The year demographic data collection ended.
	end_month	The month demographic data collection ended.
	periodicity	Frequency of the model (1: annual transition, 2: semi-annual transition, 0.2: 5 year transition).
Spatial Metadata	population_name	The name of the population given in the data source.
	number_populations	The number of populations that a given model describes.
	lat	The decimal latitude of the population.
	lon	The decimal longitude of the population.
	altitude	The altitude of the population above sea level, obtained either from the publication or Google Earth.
	country	The ISO3 code for the country or countries in which the data were collected.
	continent	The continent or continents on which the data were collected.
	ecoregion	The terrestrial or aquatic ecoregion corresponding to the population.

30. Kousta, S., Pastrana, E., & Swaminathan, S. (2019). Three approaches to support reproducible research. *Science Editor* 42(3): 77-82.
31. Jones, O.R., Scheuerlein, A., Salguero-Gómez, R., Camarda, C.G., Schaible, R., Casper, B.B., *et al.* (2014). The diversity of ageing across the tree of life. *Nature* 505: 169-173. <https://doi.org/10.1038/nature12789>
32. Jones, O.R., Barks, P., Stott, I., James, T.D., Levin, S.C., Petry, W.K. *et al.* (2021). Rcompadre and Rage - two R packages to facilitate the use of the COMPADRE and COMADRE databases and calculation of life history traits from matrix population models. *bioRxiv*. <https://doi.org/10.1101/2021.04.26.441330>

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

Synthesis

Acknowledgements

I would not, nor could not, have done this work alone. I first want to thank my supervisors, Tiffany Knight and Roberto Salgeuro-Gomez. Your mentorship has been invaluable, and I cannot think of two more supportive advisors in academia. I also want to thank the members of the Knight Lab over the last 6 years. None of these ideas would have happened were it not for the unrelenting support, intellectual brilliance, and comic relief you provided. So thanks to Aldo Compagnoni, Sanne Evers, Neeraja Venkataraman, Valentin Stefan, Dylan Craven, Joanne Bennett, Leana Zoller, Neeraja Venkataraman, Elena Motivans, Martin Andrzejak, Amibeth Thompson, Demetra Rakosy, Lotte Korell, and Michael Wohlwend.

I also owe a great debt to those who kept me sane throughout this whole ordeal, particularly Andrea Pacheco, Eduardo Arle, the three Leanas, Mike, Sanne, Max, Valentin, Martin, and Amibeth. Thanks for the impromptu nights out, the parties, and the Hops nights with Franz and Stine (and to the latter for putting up with us all these years).

I would be remiss in skipping over my friends back home: Alex, Henry, Ryan, Christian, Andy, Keith, Drew and the rest of the Denver team. Home was always a reset switch for me, and it was largely because of you that it felt that way.

Speaking of home, I now have to thank my family. To my parents, Steve and Chris, and my brothers, Daniel and M - I have no words to convey how grateful I am for your love and support, and how impossible this would have been without it. I am so grateful for my grandparents Betty and Buddy Levin for their dedication to education and their support of mine, and to Robert and Lib Conner for inspiring my love of nature. I love you all. Thanks.

Appendix 1: ipmr Case Study 1

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

α_s and β_s 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.

```
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 an 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 **density-independent** IPM.
- B. **dd**: This is used to denote a **density-dependent** 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 pro-

vide 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 NAs 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
)

```

```

asypg_grow_rate <- lambda(carpobrotus_ipm)
asypg_grow_rate

```

```
## [1] 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).
```

```
asypg_grow_rate - lam_eigen
```

```
## [1] 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 diagnostic 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)

## [1] 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.

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)

```

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)
}
```

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_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).

```

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 return a list with `d_dbh` and `d_height`.

```

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
```

```
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))

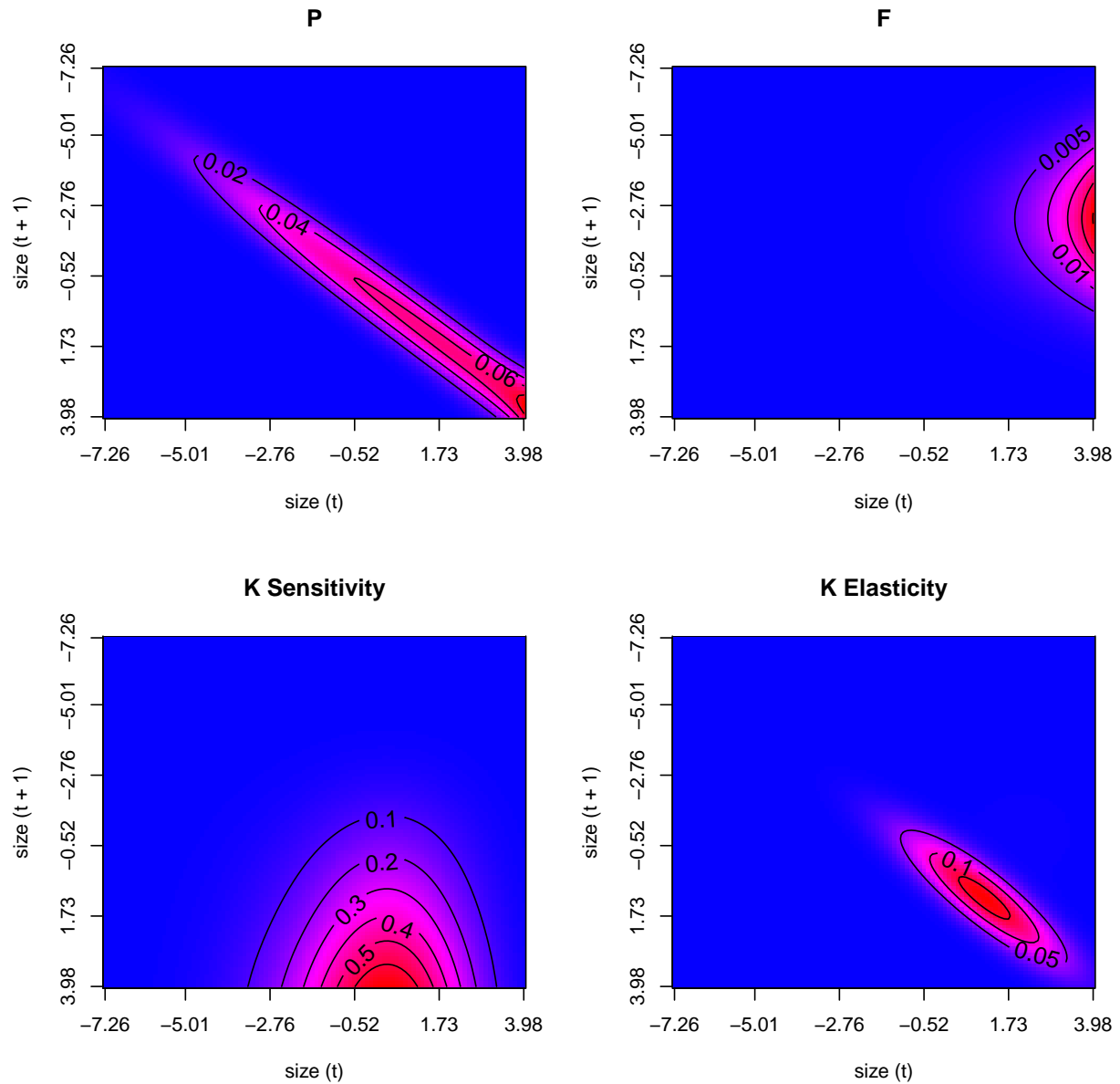
# Sensitivity and elasticity

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))

```

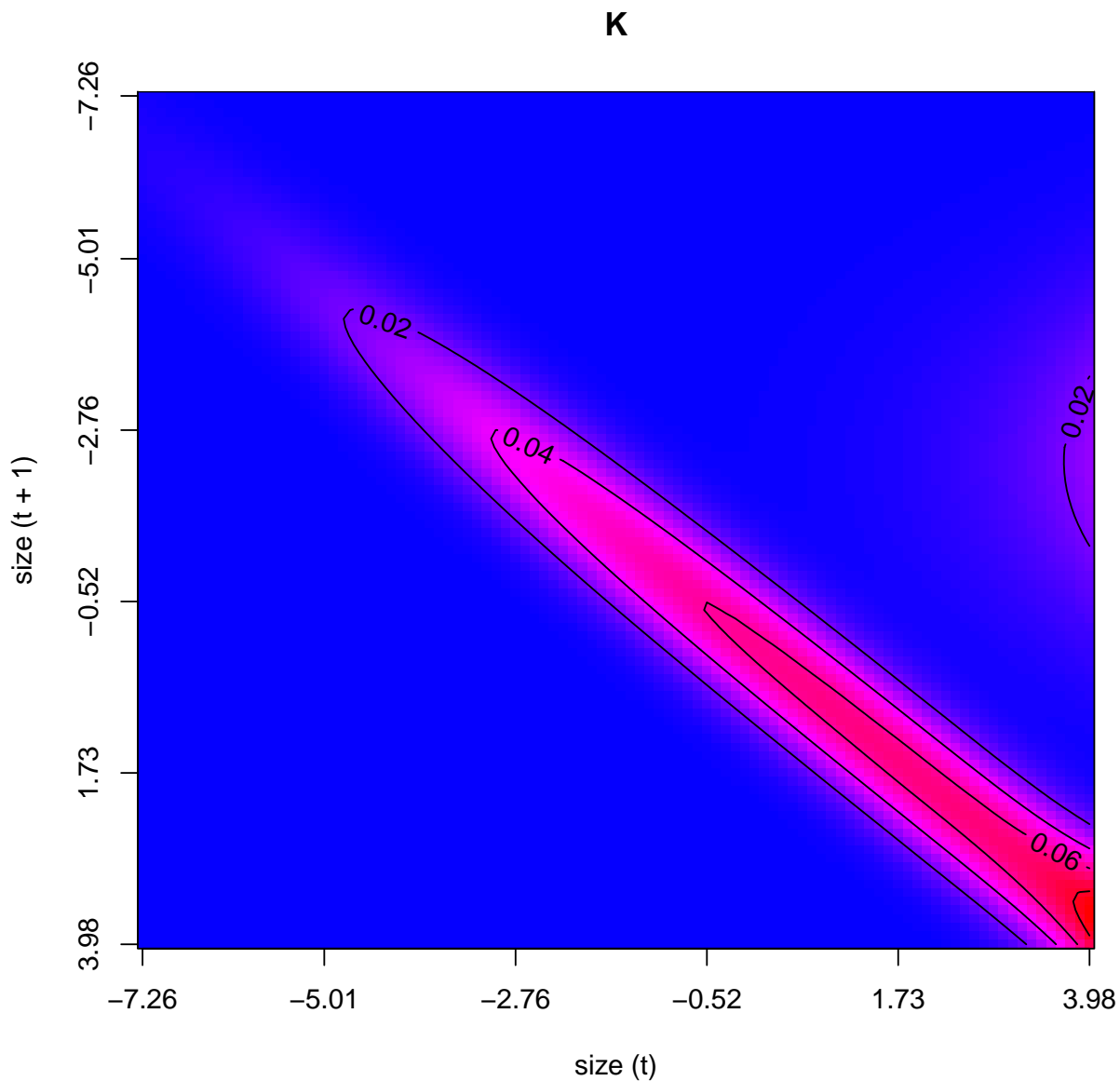


If we want to plot the iteration kernel, we can use `ipmr`'s `make_iter_kernel()` function to create one, and then the `plot()` method to plot that as well.

```
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")

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),

```

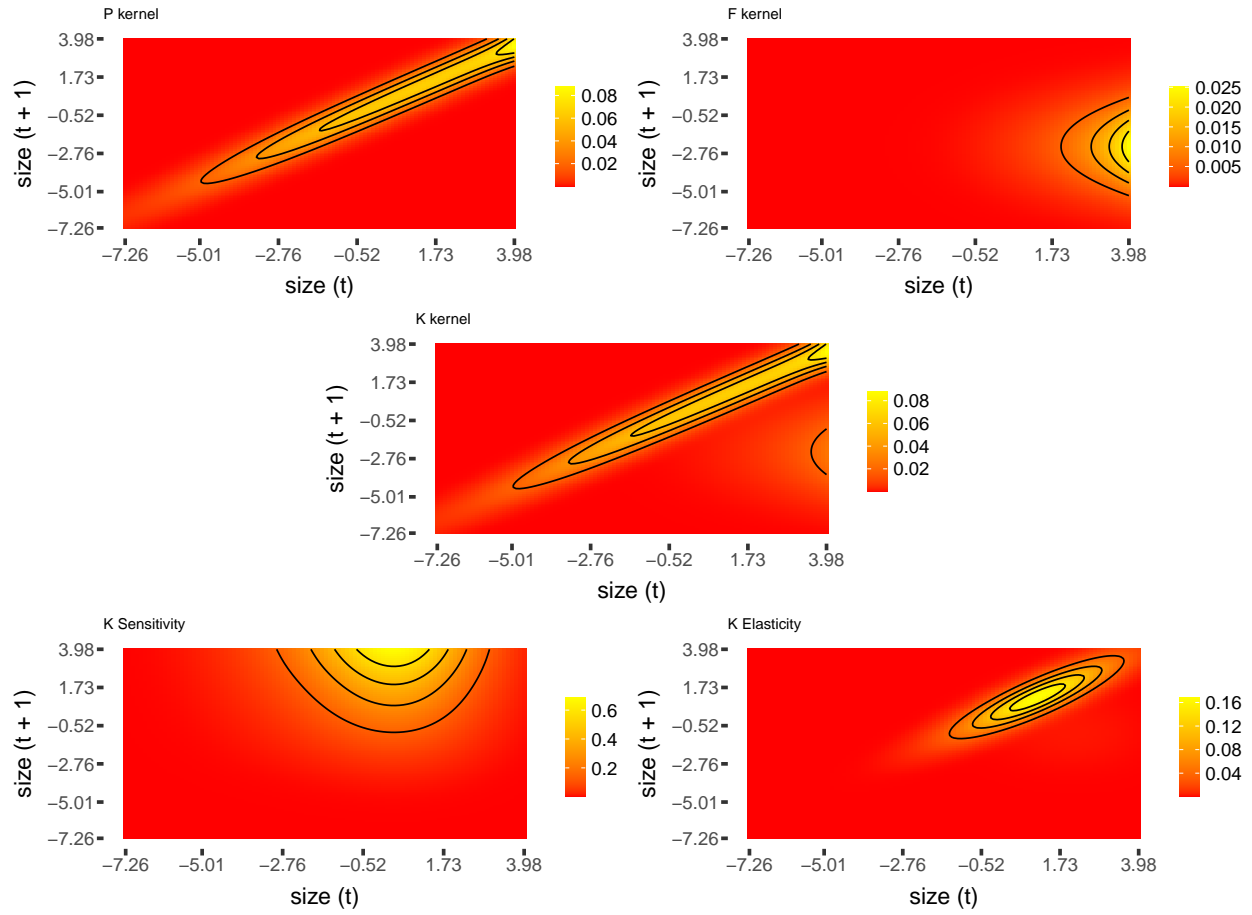
```

        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))

```



Appendix 2: ipmr Case Study 2

Case Study 2: Ellner, Childs & Rees 2016

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

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: $\text{Logit}(s(z, a)) = \alpha_s + \beta_{s,z} * z + \beta_{s,a} * a$
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: $\text{Logit}(p_b(z, a)) = \alpha_{p_b} + \beta_{p_b, z} * z + \beta_{p_b, a} * a$

11. Probability of recruiting: $\text{Logit}(p_r(a)) = \alpha_{p_r} + \beta_{p_r, a} * a$

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×size IPM. We are going to skip the simulation and regression model fitting steps and just use the “true” parameter estimates to generate the IPM.

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(repr_int + repr_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
  )

```

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

```

```
## [1] 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 $n \times n$ kernel, rather than a $1 \times n$ 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)
```

```
## [1] 1.017868
```

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

Further analyses

The `right_ev` and `left_ev` functions also work for $\text{age} \times \text{size}$ 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)

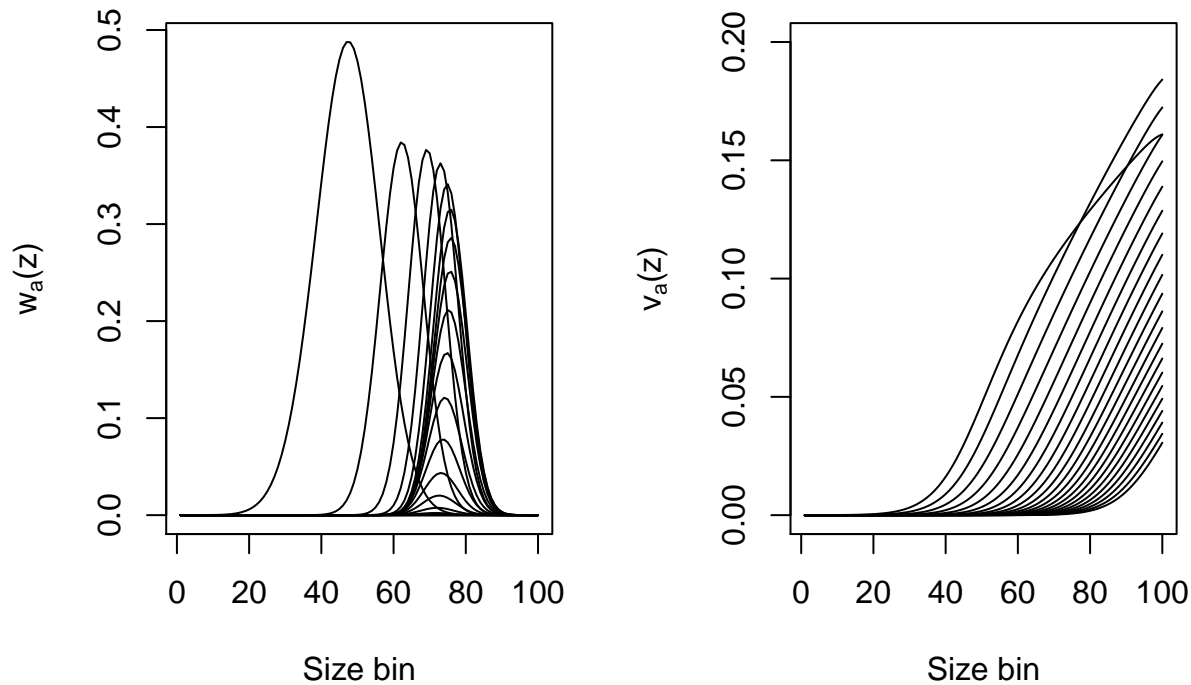
v_plot <- lapply(repro_values, function(x, d_z) x / d_z,
                 d_z = d_z)

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
}

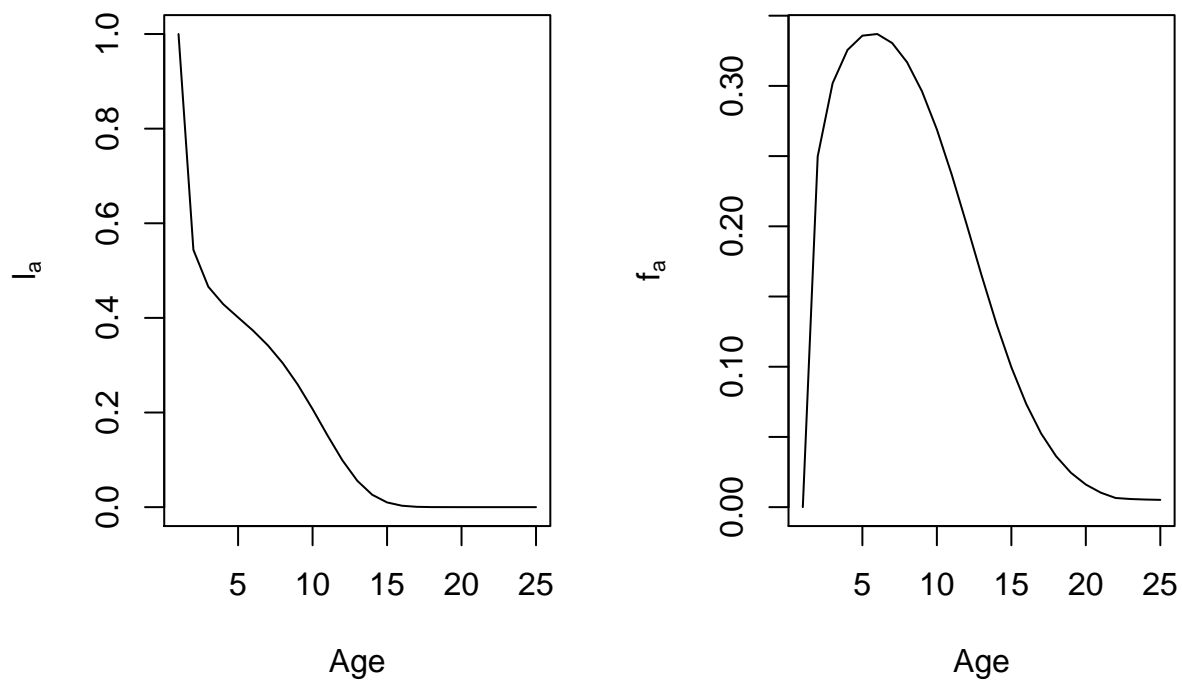
f_a <- f_a / l_a

# 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")

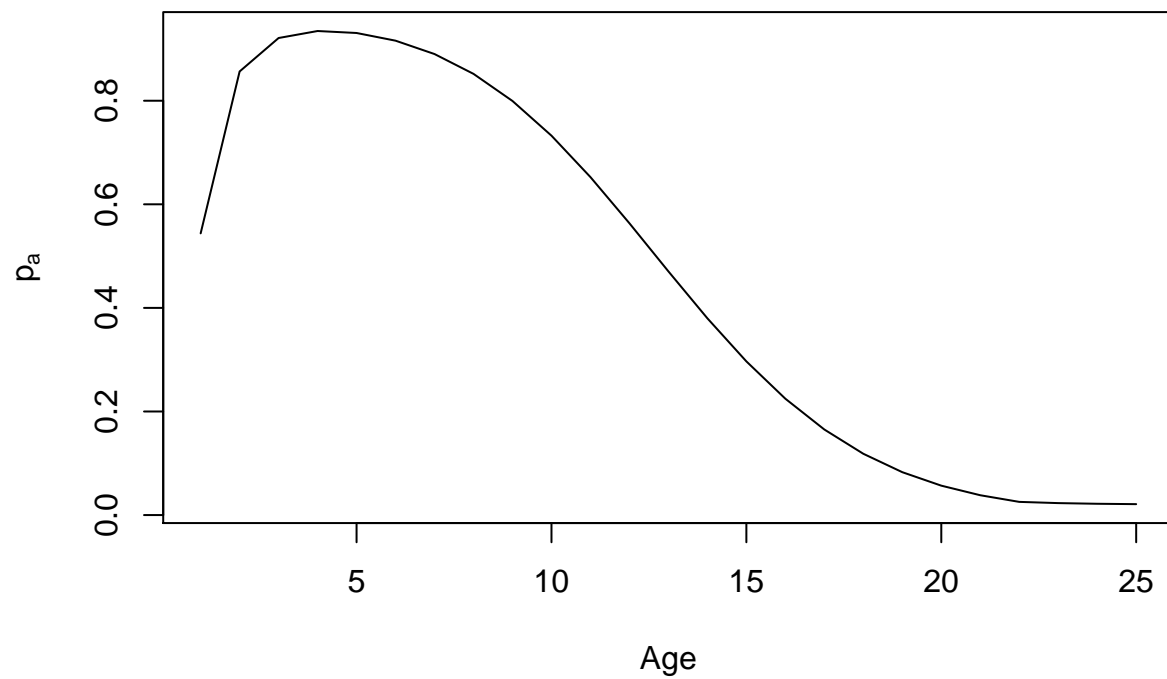
```



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")
```



Citations

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.

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

Supplementary Information for Chapter 2

Figure S3.1

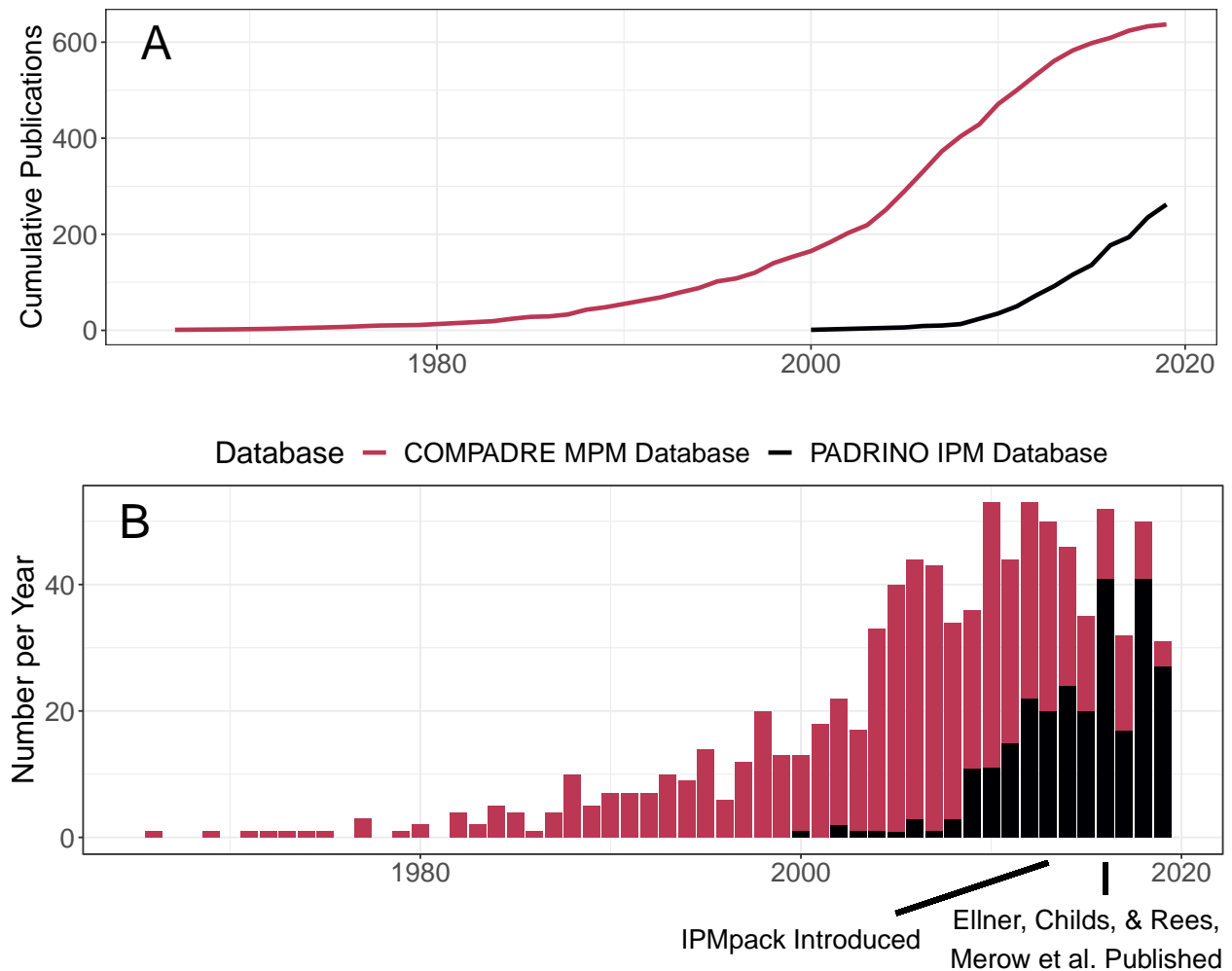


Figure S3.1: The usage of integral projection models (IPMs) has increased rapidly since their introduction. Cumulative number of publications using matrix projection models (MPMs, red) and IPMs (black) (A) and number of publications per year for each type of model (B). IPMs have been adopted rapidly 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.

Appendix 4: PADRINO Case Study 1

PADRINO and Rpadrino

We have created *Rpadrino* to streamline the process of interacting with PADRINO from *R*. The goal of *Rpadrino* is data management and model construction - not necessarily to do analyses for you. Thus, there is still some minimum amount of programming knowledge required to use it. It will also be helpful to understand how *ipmr*, the engine that powers model reconstruction, creates model objects and the things that it returns. *ipmr* is extensively documented [here](#), and reading at least the introduction to that will certainly help understand the code that follows here. Eventually, we plan to create the *ipmtools* package, which will house functions designed to work with the IPMs stored in PADRINO to conduct more extensive analyses (e.g. perturbations, LTREs, life history traits).

Usage

This case study makes use of *dplyr* to help with data transformation. If you do not already have it, install it (and *Rpadrino*) with:

```
install.packages(c("dplyr", "Rpadrino"))
```

We will show how to compute sensitivity and elasticity for simple models, and then derive some demographic quantities from models housed in PADRINO. The first step is to identify models that have the information we want. Sensitivity and elasticity computations will make use of functions contained in *ipmr*, and we will define a couple of our own to help tie it all together.

After perturbation analyses, we will compute the mean lifetime output of recruits as a function of initial size z_0 , $\bar{r}(z_0)$. This is defined as $\bar{r}(z_0) = eFN$, where F represents a the fecundity kernel, and N is the fundamental operator. The fundamental operator can be thought of as the expected amount of time spent in any state z' prior to death given an initial state z_0 (Caswell 2001), and is computed as $N = (I - P)^{-1}$ (where P is the survival/growth kernel, and I is an identity operator such that $IP = PI = P$).

Finally, we will compute mean size at death ($\bar{\omega}(z_0) = (\mathbf{i} \circ (1 - s))N$), and the size at death kernel ($\Omega(z', z_0) = (1 - s(z'))N(z', z_0)$). Thus, we need models that contain information on survival and growth, and sexual reproduction. To keep things simpler, we will restrict ourselves to simple IPMs.

NB: The above formulae are from Ellner, Childs, & Rees (2016), Table 3.3. Their derivations are described in detail in Chapter 3 of the book.

Subsetting using Metadata and other tables

We can find simple IPMs in PADRINO using a combination of *dplyr* (Wickham et al. 2021) and *Rpadrino* code. *Rpadrino* provides the `pdb_subset()` function. `pdb_subset()` currently only takes `ipm_ids` that we want to keep. The functionality will get expanded, but it's surprisingly complicated to manage that in a user-friendly interface (see the [PADRINO explorer app](#) for additional help). This means that we have to work out which `ipm_ids` correspond to the models we want, and then pass those to `pdb_subset()`.

It is a good idea to consult the [table guide](#) so that you are familiar with table and variable names, and what information they provide. This case study will introduce some of the variables and tables, but it does not cover them all!

```
library(dplyr)
library(Rpadrino)

pdb <- pdb_download(save = FALSE)
```

```
# Simple models only make use of 1 trait/state variable. Therefore, if a model
# has more than 1, it is, by definition, not a simple model. The code below
# calculates the number of traits per "ipm_id", and then filters out those
# that have more than 1 trait. The final piece with the square brackets makes sure
# the final result is a character vector containing only ipm_id's, rather than
# a data.frame
```

```
simple_mod_ind <- pdb$StateVariables %>%
  group_by(ipm_id) %>%
  summarise(N = n()) %>%
  filter(N < 2) %>%
  .[, 1, drop = TRUE]

simple_pdb <- pdb_subset(pdb, simple_mod_ind)
```

We have quite a few to choose from! However, a number of these may be stochastic and/or density-dependent models. We will want to get rid of those too, as sensitivity analyses can be trickier and more time consuming for them. This process will use the `Metadata`, `EnvironmentalVariables`, and `ParSetIndices` tables to find those.

```
# The first piece of stoch_ind examines the EnvironmentalVariables table. This
# contains information on IPMs that include continuous environmental variation.
# Rpadrino treats these as stochastic by default, because PADRINO almost
# always uses random number generators to sample the distributions of environmental
# values. Therefore, there is not really a way to sample these in a way that makes
# them deterministic while still preserving the published model.
```

```
stoch_ind <- unique(simple_pdb$EnvironmentalVariables$ipm_id)
```

```
# The second piece of stoch_ind examines the ParSetIndices table. This table
# describes discrete environmental variation. These models do not have to be
# stochastic, but they will make the analysis a bit more complicated, so we are
# going to drop those for now.
```

```
stoch_ind <- c(stoch_ind, unique(simple_pdb$ParSetIndices$ipm_id))
```

```
# The final piece of stoch_ind checks for density dependence. This information
# is stored in the 'has_dd' column of the Metadata table.
```

```
stoch_ind <- c(stoch_ind,
  unique(simple_pdb$Metadata$ipm_id[simple_pdb$Metadata$has_dd]))
```

```
det_pdb <- pdb_subset(simple_pdb, setdiff(simple_pdb$Metadata$ipm_id,
  stoch_ind))
```

For simple models in PADRINO, the kernels are pretty consistently named with respect to the broader IPM literature: P denotes survival and growth, F denotes sexual reproduction, and C denotes asexual reproduction. The `IpmKernels` table stores the names, functional forms of the kernels, as well as other information needed to implement them. We can use the kernel names column, `kernel_id`, to do a quick sanity check to make sure our subsetting produced only these kernels like so:

```
unique(det_pdb$IpmKernels$kernel_id)
```

```
## [1] "P" "F"
```

Great, all Ps and Fs! Finally, we are going to make sure we only have one IPM per species in our analysis.

We can do this using the `Metadata` table. This is certainly not required for any analyses, just to keep things tractable for now.

```
keep_ind <- det_pdb$Metadata$ipm_id[!duplicated(det_pdb$Metadata$species_accepted)]

my_pdb <- pdb_subset(det_pdb, keep_ind)
```

Re-building IPMs

Now that we have our data subsetting, we can start making IPMs. The first step is always to create `proto_ipm` objects. These are an intermediate step between the database and a set of usable kernels. Because *ipmr* also uses these as an intermediate step, we can combine models from PADRINO with ones that we create ourselves. There is an example of this in the second case study.

Under the hood

If you're interested in how PADRINO actually stores IPMs, and specifically the expressions that comprise them, keep reading. If not, skip to the next heading.

Lets have a look at how PADRINO stores kernels, vital rates, and parameters. These are in the `IpmKernels`, `VitalRateExpr`, and `ParameterValues` tables, respectively.

```
head(my_pdb$IpmKernels[ , 1:3])
```

##	ipm_id	kernel_id	formula
## 43	aaaa34	P	$P = s * g * d_lnsize$
## 44	aaaa34	F	$F = r * fn * pE * d * d_lnsize$
## 47	aaaa36	P	$P = s * g * d_lnsize$
## 48	aaaa36	F	$F = r * fn * pE * d * d_lnsize$
## 92	aaa144	P	$P = s * g * d_size$
## 93	aaa144	F	$F = rep_p * es_p * sdl_s * n_infl * n_fl * n_seed * d_size$

```
head(my_pdb$VitalRateExpr[ , 1:3])
```

##	ipm_id	demographic_parameter	formula
## 147	aaaa34	Survival	$s = 1/(1+\exp(-(s_b + s_m * lnsize_1)))$
## 148	aaaa34	Growth	$g = \text{Norm}(g_mean, g_var)$
## 149	aaaa34	Growth	$g_mean = g_b + g_m * lnsize_1$
## 150	aaaa34	Growth	$g_var = \text{sqrt}(gv_b + gv_m * lnsize_1)$
## 151	aaaa34	Fecundity	$r = 1/(1+\exp(-(r_b + r_m * lnsize_1)))$
## 152	aaaa34	Fecundity	$fn = \exp(fn_b + fn_m * lnsize_1)$

```
head(my_pdb$ParameterValues)
```

##	ipm_id	demographic_parameter	state_variable	parameter_name	parameter_value
## 619	aaaa34	Survival	lnsize	s_b	-0.5612335
## 620	aaaa34	Survival	lnsize	s_m	0.4628431
## 621	aaaa34	Growth	lnsize	g_b	1.1088198
## 622	aaaa34	Growth	lnsize	g_m	0.5148672
## 623	aaaa34	Growth	lnsize	gv_b	0.9504887
## 624	aaaa34	Growth	lnsize	gv_m	0.0000000

We can see that the kernels and vital rate expressions are all defined symbolically, and the parameter values are stored elsewhere. This helps us reuse parameters that appear in multiple expressions without re-typing them, reducing the risk of errors. Additionally, it'll make it easier for us to modify parameter values, vital rate expressions, and kernel formulae if we want to. However, the syntax in the tables is probably not the

easiest to work with directly. Therefore, *Rpadrino* provides the `pdb_make_proto_ipm()` function. This takes a `pdb` object and produces a list of `proto_ipms`. In the chunk after this one, we will see that it translates the syntax in `IpmKernels` and `VitalRateExpr` into usable R code. There are additional options that we can pass to this, but we will ignore those for now, and just focus on creating and understanding what the outputs are.

Creating the `proto_ipm` list

The following line generates a set of `proto_ipm`'s for the species in our subsetting database:

```
simple_det_list <- pdb_make_proto_ipm(my_pdb)

## 'ipm_id' aaa310 has the following notes that require your attention:
## aaa310: 'Geo and time info retrieved from COMPADRE (v.X.X.X.4)'

## 'ipm_id' aaa323 has the following notes that require your attention:
## aaa323: 'Simulated demographic data derived from Nicole J Ecol 2011'

## 'ipm_id' aaa326 has the following notes that require your attention:
## aaa326: 'Demographic data from Metcalf Funct Ecol 2006'

## 'ipm_id' aaa385 has the following notes that require your attention:
## aaa385: 'Same data as AAA385. State variable Height (Cm)'

## 'ipm_id' ddddd3 has the following notes that require your attention:
## ddddd3: 'Frankenstein IPM'

## 'ipm_id' ddddd4 has the following notes that require your attention:
## ddddd4: 'assumes mean surface temp of 10.34 °C, and constant survival probability of
## large pike'

## 'ipm_id' dddd24 has the following notes that require your attention:
## dddd24: 'Assumes no external recruitment'

## 'ipm_id' dddd26 has the following notes that require your attention:
## dddd26: '1 ipm digitized, additional ipms taking into account dispersal still
## possible to digitize'

## 'ipm_id' dddd30 has the following notes that require your attention:
## dddd30: 'Frankenstein IPM'

## 'ipm_id' dddd37 has the following notes that require your attention:
## dddd37: 'MS contains 2 det and 2 stoch IPMs, only 1 det included here'

## 'ipm_id' dddd39 has the following notes that require your attention:
## dddd39: 'Only deterministic model included here: assumes precipitation = 104mm'

## 'ipm_id' dddd40 has the following notes that require your attention:
## dddd40: 'DEB-IPM - these vital rates assume NO shrinking. 1 model digitized: assumes
## that scaled functional response E_Y = 0.65, with var(E_Y) = 0.1 - see paper for
## details'

## 'ipm_id' dddd41 has the following notes that require your attention:
## dddd41: 'DEB-IPM - these vital rates shrinking IS possible. 1 model digitized:
## assumes that scaled functional response E_Y = 0.65, with var(E_Y) = 0.1 - see paper
## for details'
```

First, we note that the building process threw out a few messages. The first is that the coordinates and duration information come from COMPADRE, not necessarily the original publication. This is not really alarming - COMPADRE is pretty trustworthy. The next few are related to demographic data sources and GPS location. “Frankenstein IPM” refers to a situation where some vital rates are measured directly from

demographic data the authors collected, while other vital rates were retrieved from the literature (*i.e.* the object is cobbled together from disparate sources, Shelley 1818). Again, not necessarily alarming, though we'd want to know that if our study question required that all vital rates come from one place (*e.g.* matching environmental conditions to demographic performance). The next few tell us that the publication actually contains more IPMs, but that our PADRINO digitization team hasn't finished entering all of them yet. And finally, there is a note about the assumptions contained by the model.

we will inspect a couple of the objects in this list to get a feel for what a `proto_ipm` contains:

```
simple_det_list
```

```
## This list of 'proto_ipm's contains the following species:
```

```
## Poa alsodes
## Poa sylvestris
## Aeonium haworthii
## Cotyledon orbiculata
## Aconitum noveboracense
## Dracocephalum austriacum
## Cirsium arvense
## Lonicera maackii
## Mimulus cardinalis
## Reynoutria japonica
## Carpobrotus spp
## Crocodylus niloticus
## Esox lucius
## Sisturus catenatus catenatus
## Ovis aries
## Oncorhynchus clarkii
## Tridacna maxima
## Gadus morhua
## Podarcis lilfordi
## Nerodia sipedon
## Ostrea edulis
## Dipsastraea favus
## Platygyra lamellina
## Ficedula hypoleuca
## Testudo graeca
## Manta alfredi
## Rhizoglyphus robini
##
```

```
## You can inspect each model by printing it individually.
```

```
simple_det_list$aaaaa34
```

```
## A simple, density independent, deterministic proto_ipm with 2 kernels defined:
```

```
## P, F
```

```
##
```

```
## Kernel formulae:
```

```
##
```

```
## P: s * g
```

```
## F: r * fn * pE * d
```

```
##
```

```
## Vital rates:
```

```
##
```

```
## s: 1/(1 + exp(-(s_b + s_m * lnsize_1)))
```

```
## g_mean: g_b + g_m * lnsize_1
```

```

## g_var: sqrt(gv_b + gv_m * lnsize_1)
## g: stats::dnorm(lnsize_2, g_mean, g_var)
## r: 1/(1 + exp(-(r_b + r_m * lnsize_1)))
## fn: exp(fn_b + fn_m * lnsize_1)
## d: stats::dexp(lnsize_2, 1/d_mean)
##
## Parameter names:
##
## [1] "s_b"      "s_m"      "g_b"      "g_m"      "gv_b"     "gv_m"     "r_b"      "r_m"      "fn_b"     "fn_m"     "t_r"
## [13] "pE"
##
## All parameters in vital rate expressions found in 'data_list': TRUE
##
## Domains for state variables:
##
## lnsize: lower_bound = 0, upper_bound = 5, n_meshpoints = 500
##
## Population states defined:
##
## n_lnsize: Pre-defined population state.
##
## Internally generated model iteration procedure:
##
## n_lnsize_t_1: right_mult(kernel = P, vectr = n_lnsize_t) + right_mult(kernel = F,
##      vectr = n_lnsize_t)
simple_det_list$dddd30

## A simple, density independent, deterministic proto_ipm with 2 kernels defined:
## P, F
##
## Kernel formulae:
##
## P: s * g
## F: s * r * pg * 0.5 * d
##
## Vital rates:
##
## s: 1/(1 + exp(-(aS + bS * svl_1 + cS * svl_1^2)))
## muG: svl_1 + (Linf - svl_1) * (1 - exp(-k * tg))
## g: stats::dnorm(svl_2, muG, sigmaG)
## r: exp(aR + bR * svl_1)
## pg: ifelse(svl_1 < svlM, 0, 1)
## d: stats::dnorm(svl_2, muD, sigmaD)
##
## Parameter names:
##
## [1] "aS"      "bS"      "cS"      "Linf"    "k"      "tg"      "sigmaG"  "aR"      "bR"      "svlM"    "muD"
##
## All parameters in vital rate expressions found in 'data_list': TRUE
##
## Domains for state variables:
##
## svl: lower_bound = 120, upper_bound = 1200, n_meshpoints = 1000
##

```

```
## Population states defined:
##
## n_svl: Pre-defined population state.
##
## Internally generated model iteration procedure:
##
## n_svl_t_1: right_mult(kernel = P, vectr = n_svl_t) + right_mult(kernel = F,
##      vectr = n_svl_t)
```

We can see that *Rpadrino* has translated PADRINO's syntax into a set of *R* expressions that correspond to the vital rate functions and sub-kernel functional forms, as well as checked that the model can be implemented with the parameter values that are present in PADRINO. Finally, it has generated the model iteration expression, which shows how the sub-kernels interact with each trait distribution at time t to produce new trait distributions at time $t + 1$. We can now build the actual IPM objects. We will also check for convergence to asymptotic dynamics using the `is_conv_to_asymptotic` function.

```
all_ipms <- pdb_make_ipm(simple_det_list)

check_conv <- is_conv_to_asymptotic(all_ipms)
```

```
## The following IPMs did not converge: aaa310, aaa341, ccccc1, dddd3, dddd5,
## dddd10, dddd24, dddd26, dddd30, dddd33, dddd35, dddd36, dddd37, dddd39,
## dddd40, dddd41
check_conv
```

```
## [1] FALSE
```

We can see that a few of these need more than the default number of iterations to converge to asymptotic dynamics. All λ values are computed via iteration, rather than computing eigenvalues. Since we need correct λ values to compute elasticity, we will need to re-run those models until they converge (or at least come very close to convergence). `pdb_make_ipm()` contains the `addl_args` argument that tells the function how to deviate from the default behavior of `ipmr::make_ipm()`. It accepts nested lists with the following format:

```
list(<ipm_id_1> = list(<make_ipm_arg_name_1> = <XXX>,
                     <make_ipm_arg_name_2> = <YYY>),
     <ipm_id_2> = list(<make_ipm_arg_name_1> = <XXX>,
                     <make_ipm_arg_name_5> = <ZZZ>))
```

We replace the values in `<>` with the actual `ipm_ids`, argument names, and values we want them to have. We can do this many models a bit more concisely:

```
# Create an empty list with names that correspond to ipm_id's that we want to add
# additional iterations for.
```

```
ind_conv <- c(paste0("aaa", c(310,341)),
             paste0("cccc", 1),
             paste0("dddd", c(3,5)),
             paste0("ddd", c(10, 24, 26, 30, 33, 35, 36, 37, 39, 40, 41)))
```

```
# Next, we set create an entry in each list with iterations = <some number>
# we will use 250 for this example. We need to set the names of the list to be
# the ipm_id's, so that pdb_make_ipm() knows which models to use the additional
# arguments with.
```

```
arg_list <- lapply(ind_conv,
                  function(x, n_iter) list(iterations = n_iter),
                  n_iter = 250) %>%
```

```
setNames(ind_conv)

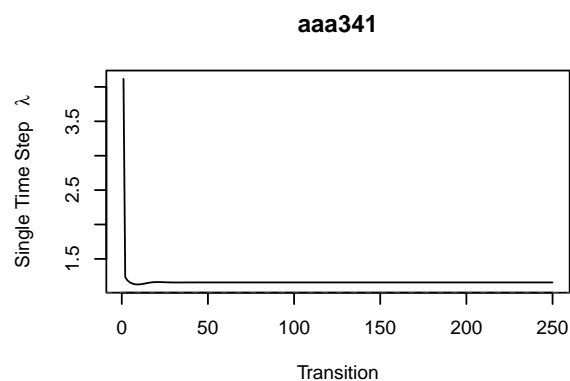
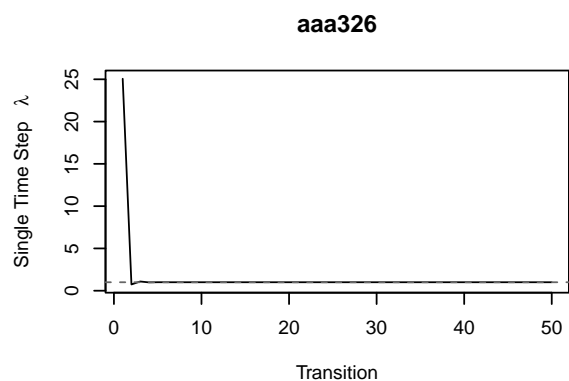
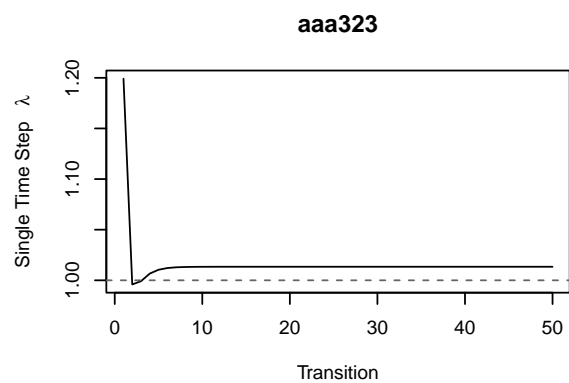
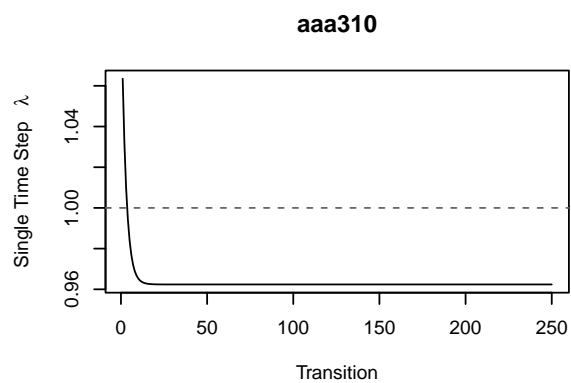
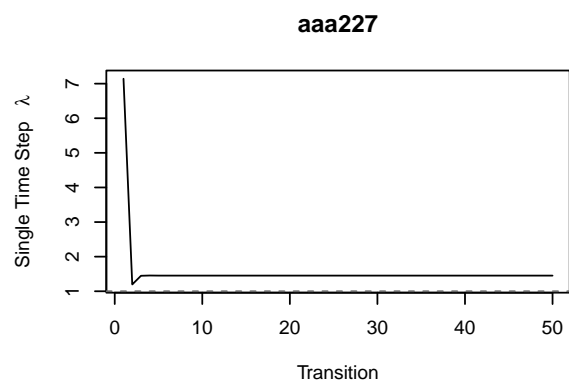
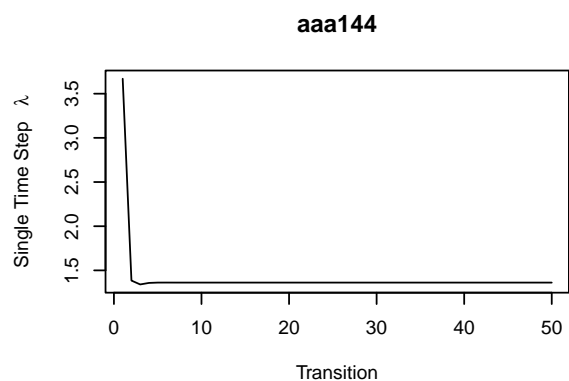
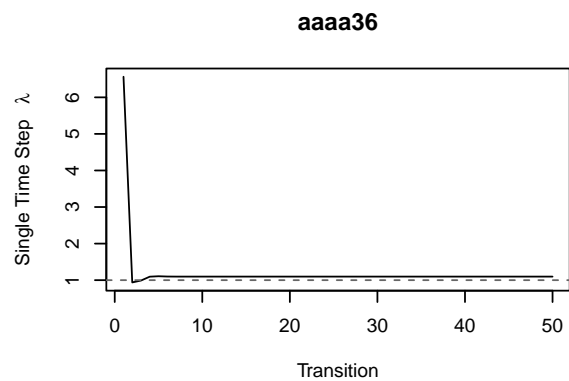
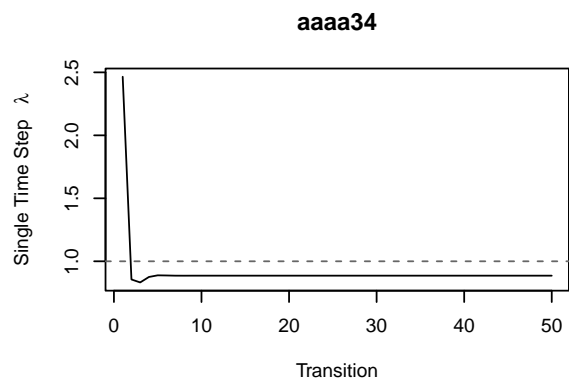
new_ipms <- pdb_make_ipm(simple_det_list, addl_args = arg_list)

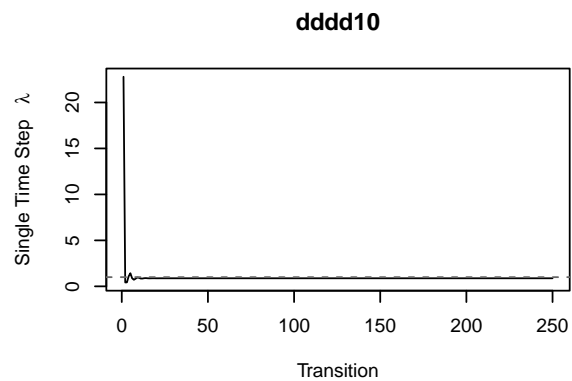
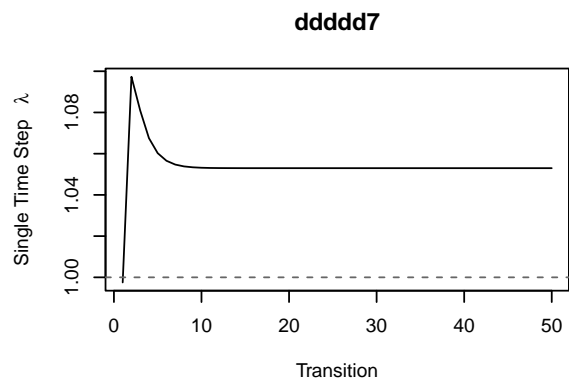
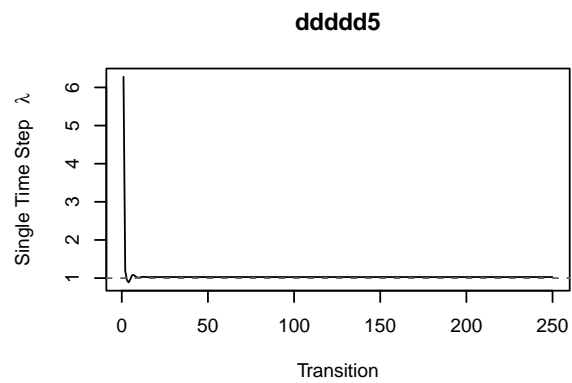
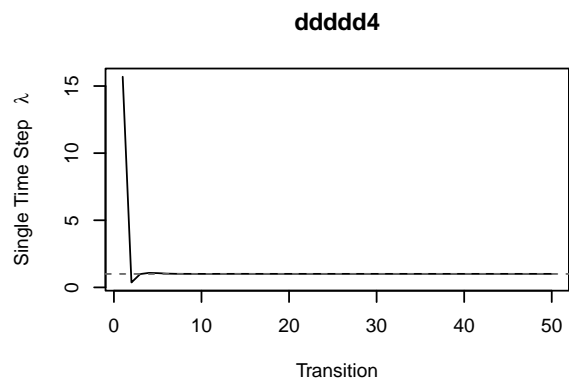
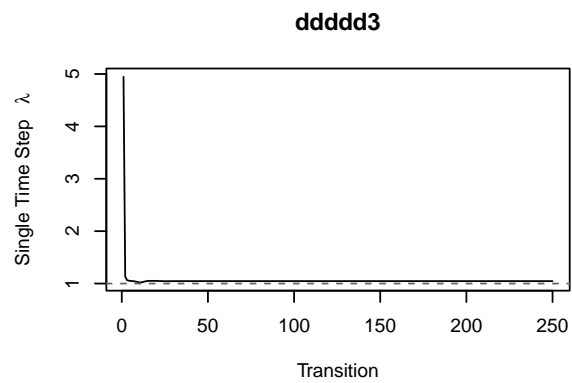
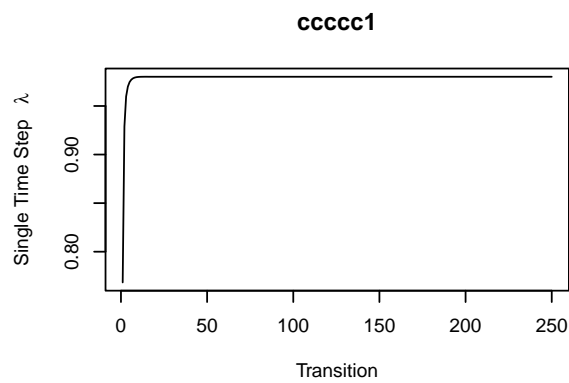
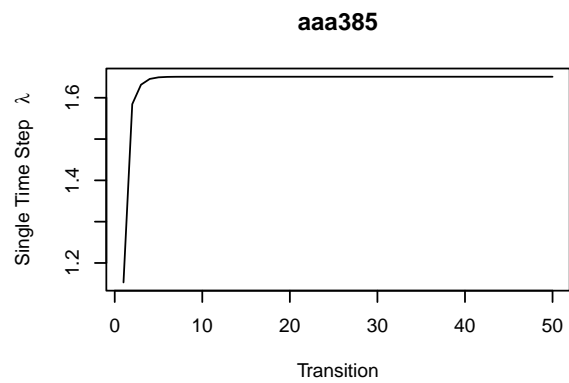
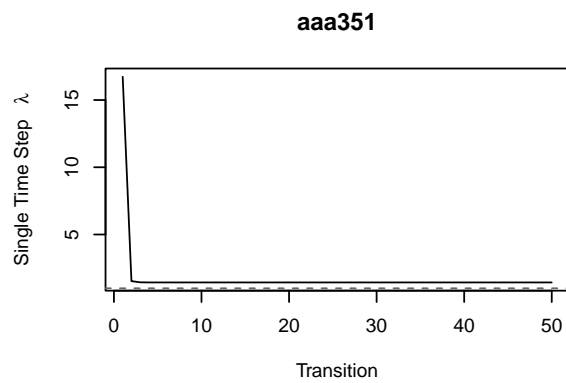
# Check for convergence out to 5 digits. This should be close enough for what
# we want to do.

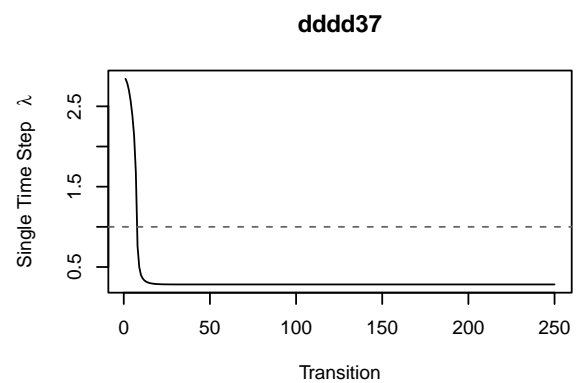
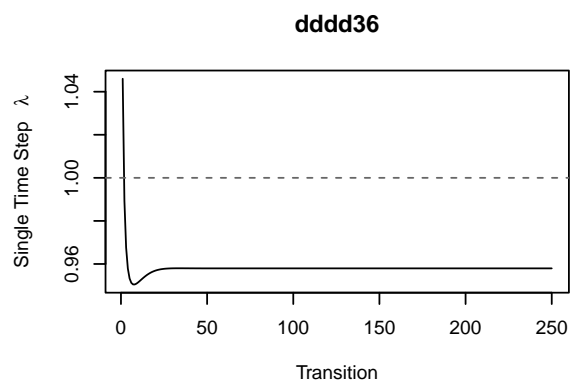
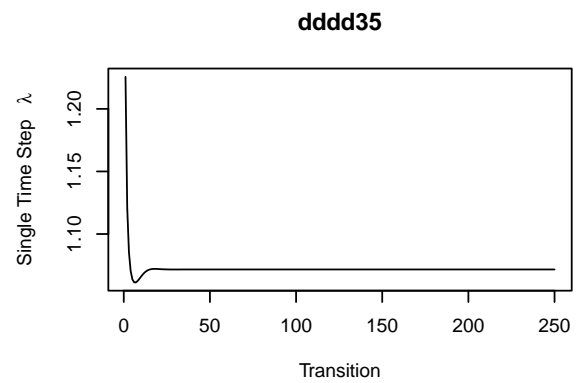
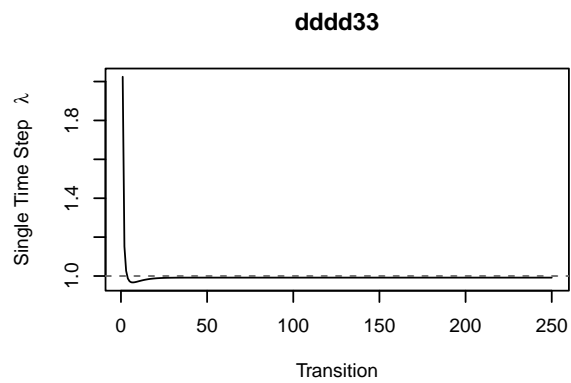
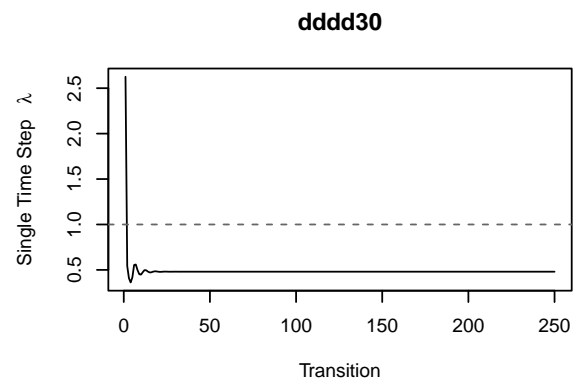
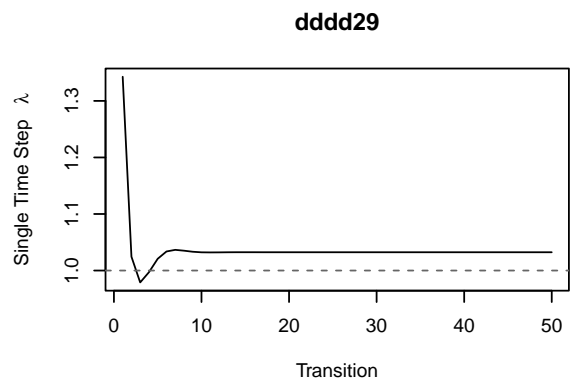
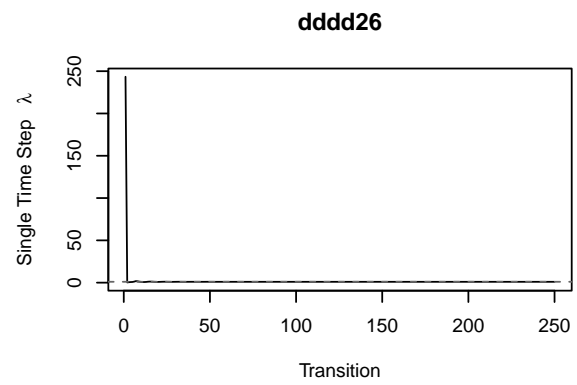
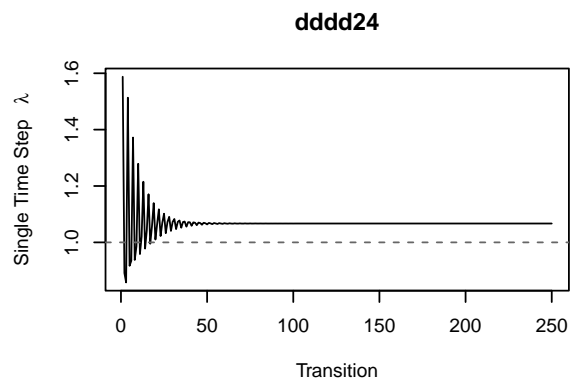
check_conv <- is_conv_to_asymptotic(new_ipms, tolerance = 1e-5)

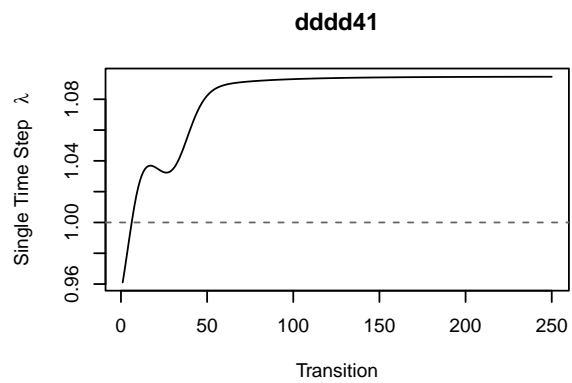
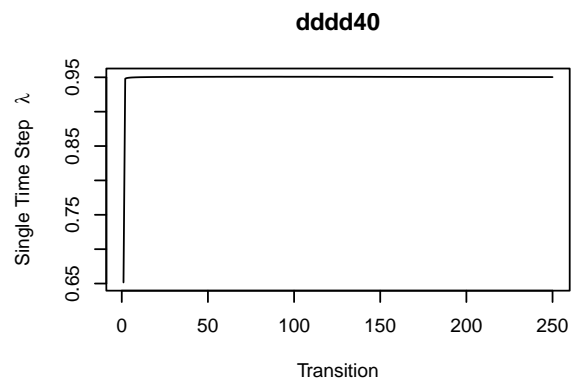
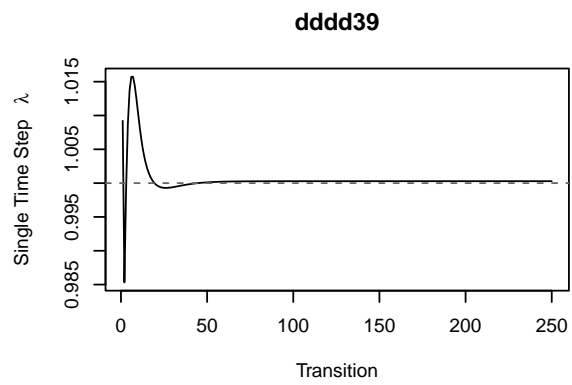
# We can also plot the lambda time series using conv_plot methods for pdb_ipms.
# The last two models may not have converged
par(mfrow = c(4, 2))

conv_plot(new_ipms)
```









dddd24, dddd26, dddd40 and dddd41 are still not converging. We will remove those from our further analyses:

```
keep_ind <- setdiff(names(new_ipms), c("dddd24", "dddd26",
                                       "dddd40", "dddd41"))

new_ipms <- new_ipms[keep_ind]
```

Further analyses

Rpadrino contains methods for most of *ipmr*'s analysis functions. These include (but are not limited to!) `lambda`, `left_ev`, and `right_ev`. We need all three of these to compute sensitivity and elasticity. We also need the binwidth of the integration mesh so we can perform integrations. *Rpadrino*'s has the `int_mesh()` function for that, and the binwidth is always the first element in the list that it returns. We can extract them like so:

```
lambdas <- lambda(new_ipms)
repro_vals <- left_ev(new_ipms, tolerance = 1e-5)
ssd_vals <- right_ev(new_ipms, tolerance = 1e-5)

d_zs <- lapply(new_ipms, function(x) int_mesh(x, full_mesh = FALSE)[[1]])
```

Sensitivity

With these, we can now compute sensitivity. This is given by $s(z'_0, z_0) = \frac{v(z'_0)w(z_0)}{\langle v, w \rangle}$, where $v(z'_0)$ is the left eigenvector and $w(z_0)$ is the right eigenvector. It will be helpful to write a function that takes these values as arguments and returns the sensitivity kernel. We will use `lapply(seq_along())` to iterate over each model.

```
# r_evs: right eigenvectors
# l_evs: left eigenvectors
# d_zs: binwidths
# lapply(seq_along()) generates a sequence of numbers that correspond to indices
# in the list of eigenvectors and binwidths. Since each of these objects is a
# list of lists, we need to use [[index]][[1]]. The first[[1]] gets the correct list
# entry, and the second [[1]] converts it to a numeric vector by unlisting the
# second layer of the list.

sens <- function(r_evs, l_evs, d_zs) {

  lapply(seq_along(r_evs),
         function(ind, r_ev, l_ev, d_z){

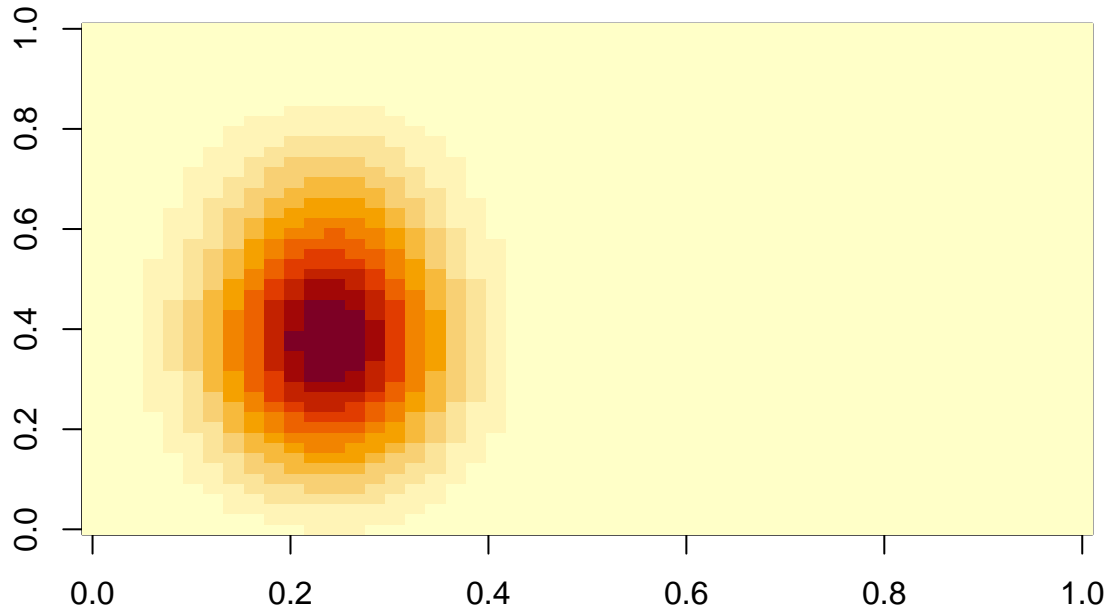
            outer(l_ev[[ind]][[1]], r_ev[[ind]][[1]]) /
              (sum(l_ev[[ind]][[1]] * r_ev[[ind]][[1]] * d_z[[ind]]))

          },
          r_ev = r_evs,
          l_ev = l_evs,
          d_z = d_zs)
}

sens_list <- sens(ssd_vals, repro_vals, d_zs) %>%
  setNames(names(lambdas))
```

We can plot these using `image()`:

```
par(mfrow = c(1, 1))
image(t(sens_list$aaa385))
```



Elasticity

We can compute elasticity without much more effort. We need one more piece of information from the IPM list that we have not extracted - the iteration kernel. We can get those using `make_iter_kernel()` on the `new_ipms` object, and then computing the elasticity using $\mathbf{e}(z'_0, z_0) = \frac{K(z'_0, z_0)}{\lambda} \mathbf{s}(z'_0, z_0)$.

```
iter_kerns <- make_iter_kernel(new_ipms)

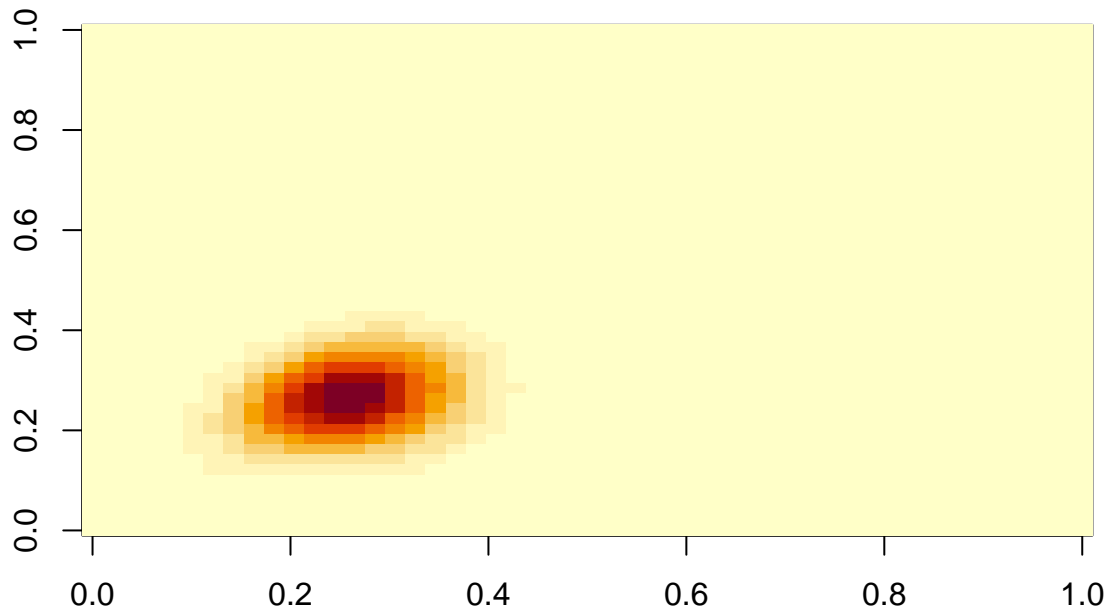
elas_list <- lapply(seq_along(iter_kerns),
  function(ind, iter_kernels, sens_kernels, lambdas, d_zs) {

    (iter_kernels[[ind]][[1]] / d_zs[[ind]] / lambdas[[ind]]) *
      sens_kernels[[ind]]

  },
  iter_kernels = iter_kerns,
  sens_kernels = sens_list,
  lambdas = lambdas,
  d_zs = d_zs) %>%
  setNames(names(lambdas))
```

Similarly, we can plot this using `image()`:

```
image(t(elas_list$aaa385))
```



Mean lifetime recruit production

We can calculate the expected number of recruits produced over an individual's lifetime as a function of its initial size. This is defined as $\bar{r}(z_0) = eFN$. F is the fecundity kernel, and we can get these kernels from each IPM in our list using:

```
F_kernels <- lapply(new_ipms, function(x) x$sub_kernels$F)
```

$N(z', z_0)$ is the fundamental operator. This tells us the expected amount of time an individual will spend in state z' given an initial state z_0 . The fundamental operator is defined as $(I - P)^{-1}$ (see Ellner, Childs, & Rees 2016 Chapter 3 for the derivation of this). I is an identity kernel ($I(z', z) = 1$ for $z' = z$, and 0 everywhere else). This code is only a bit more complicated:

```
# Function to create an identity kernel with dimension equal to P
make_i <- function(P) {
  return(
    diag(nrow(P))
  )
}

N_kernels <- lapply(new_ipms, function(x) {
  P <- x$sub_kernels$P
```

```

I <- make_i(P)

# solve() inverts the matrix for us (the ^(-1) part of the equation)
solve(I - P)

})

```

e is a constant function $e(z) \equiv 1$. In practice, the left multiplication of eF has the effect of computing the column sums of F . We will replace the e with a call to `colSums()` in our code below (this will run faster than doing the multiplication). We now have everything we need to compute and visualize the expected lifetime reproductive output:

```

# We wrap the computation in as.vector so that it returns a simple numeric vector
# rather than a 1 x N matrix

r_bars <- lapply(seq_along(N_kerns),
                 function(idx, Fs, Ns) {

                     as.vector(colSums(Fs[[idx]]) %*% Ns[[idx]])

                 },
                 Fs = F_kerns,
                 Ns = N_kerns) %>%
setNames(names(F_kerns))

# we will extract the meshpoint values so that the x-axes on our plots look
# prettier.
x_seqs <- lapply(new_ipms,function(x) int_mesh(x, full_mesh = FALSE)[[2]])

# Finally, we can plot the data by looping over the lists and creating a
# a simple line plot (type = "l")

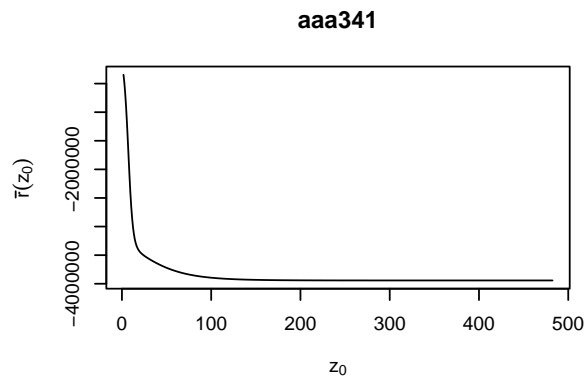
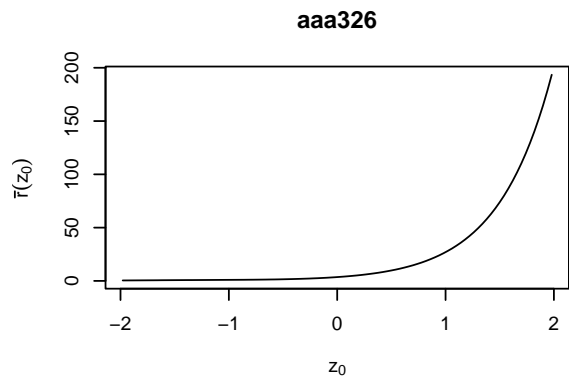
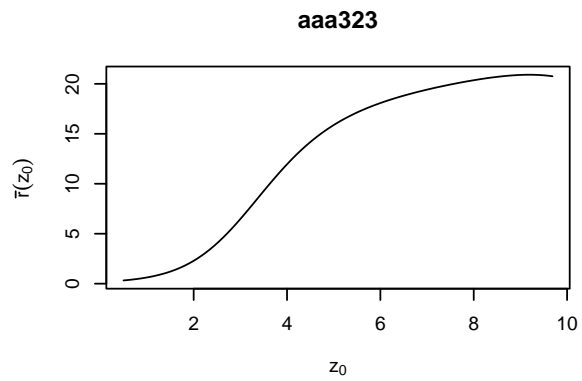
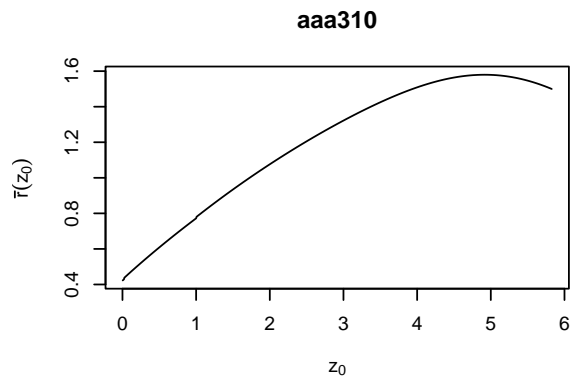
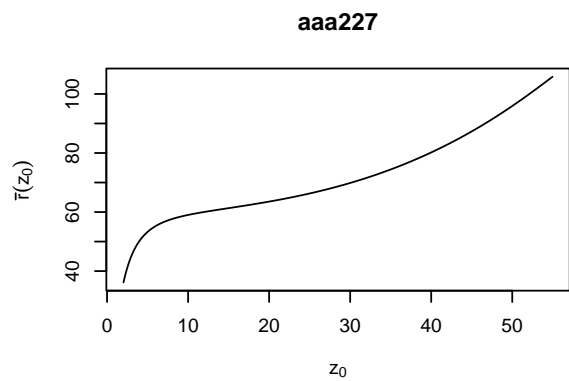
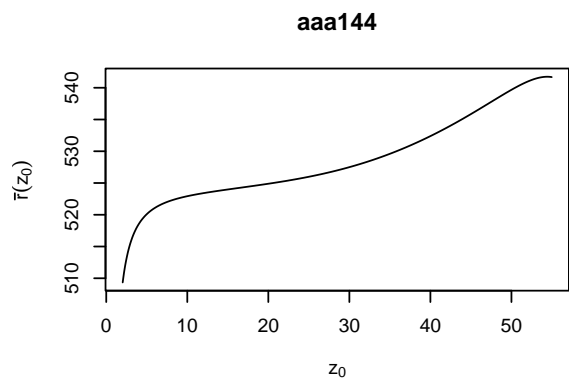
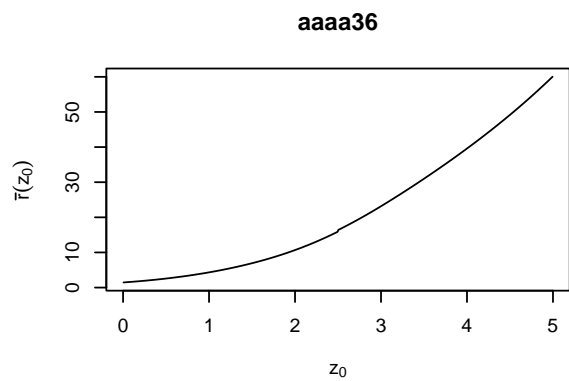
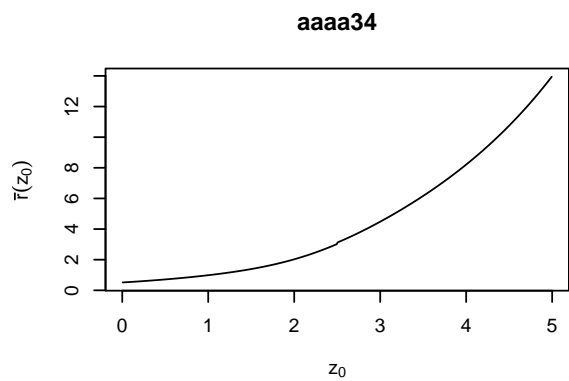
par(mfrow = c(4, 2))

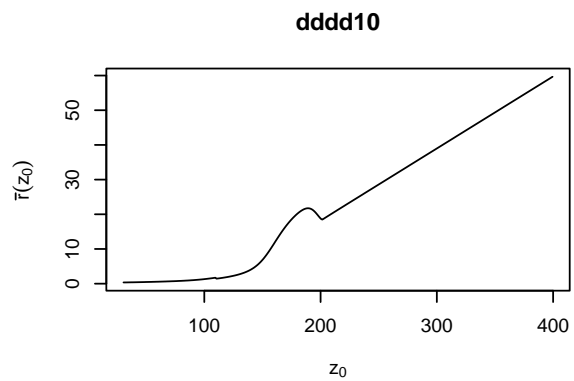
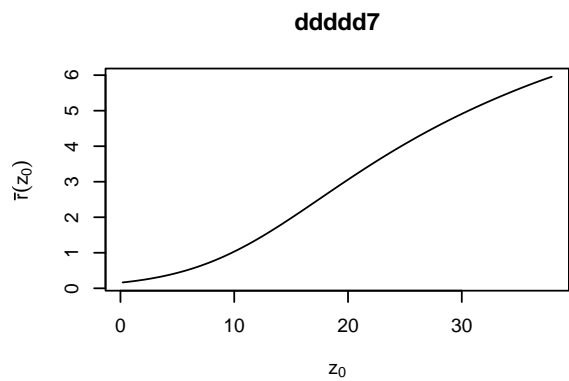
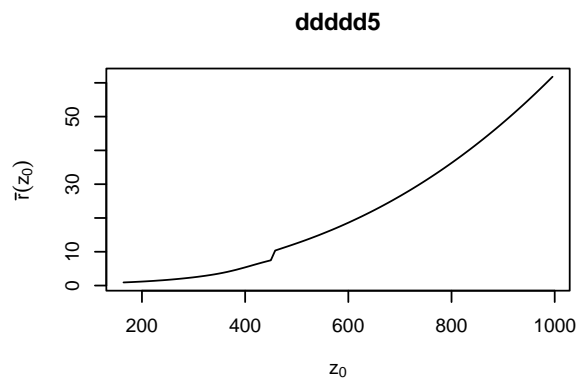
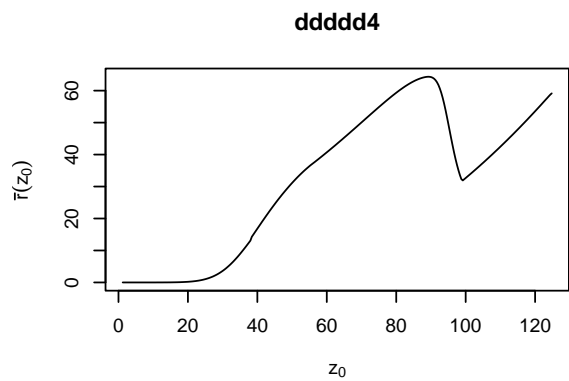
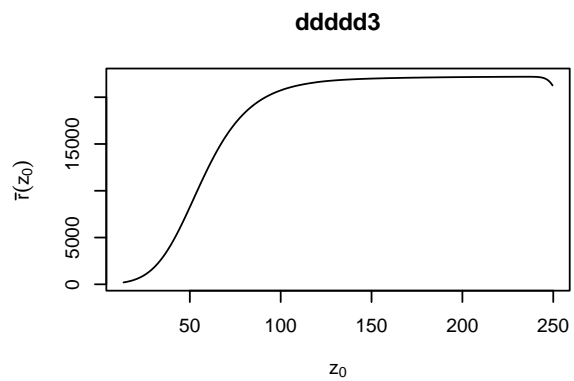
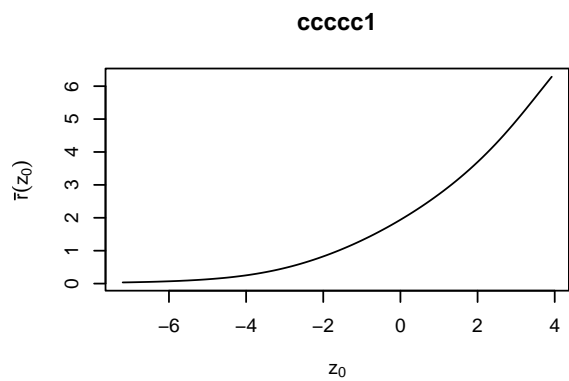
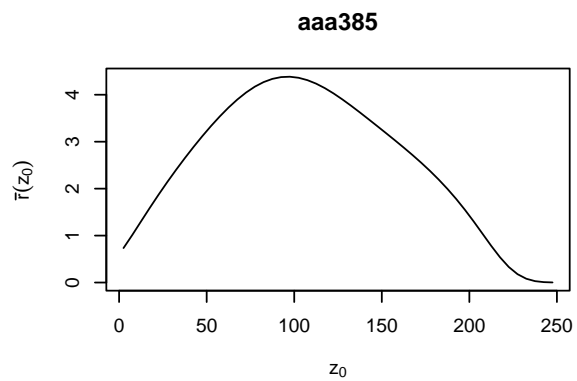
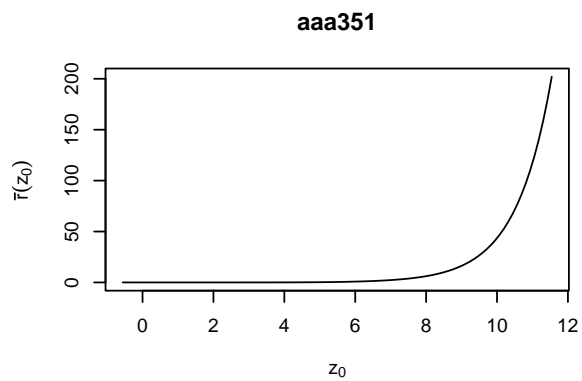
for(i in seq_along(r_bars)) {

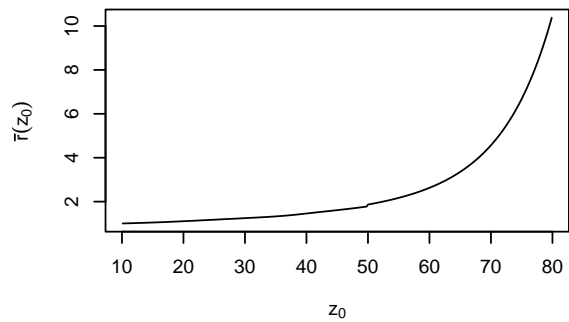
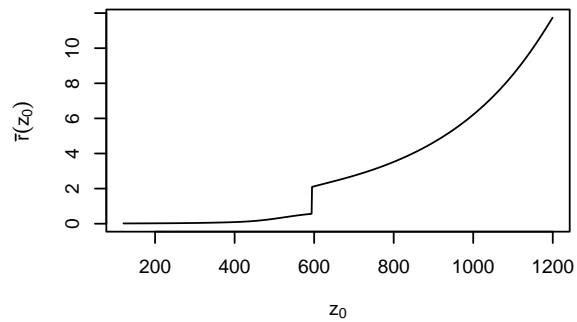
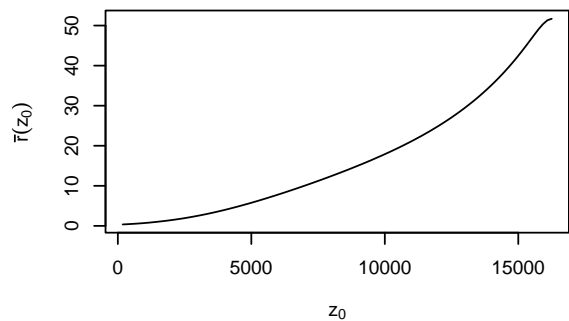
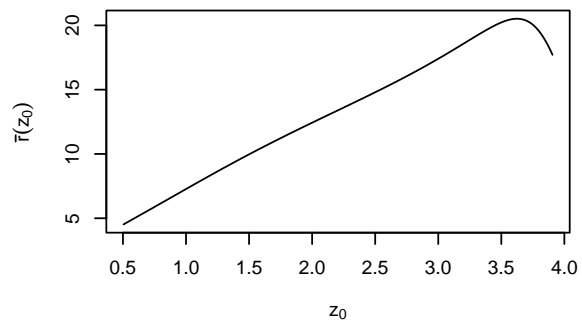
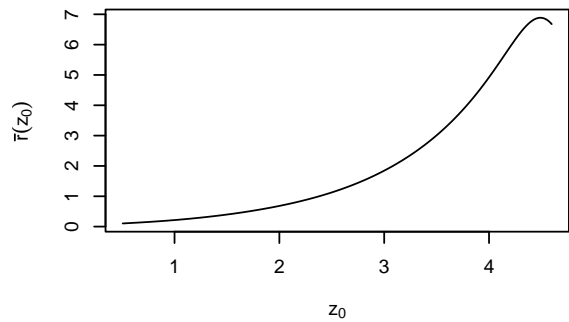
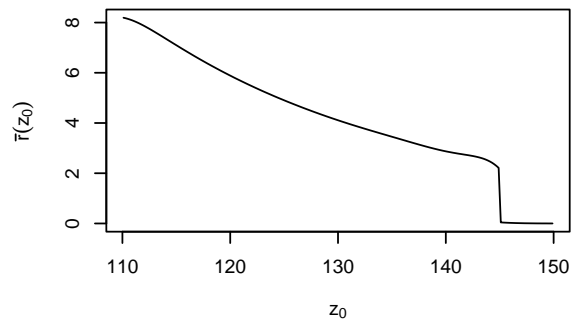
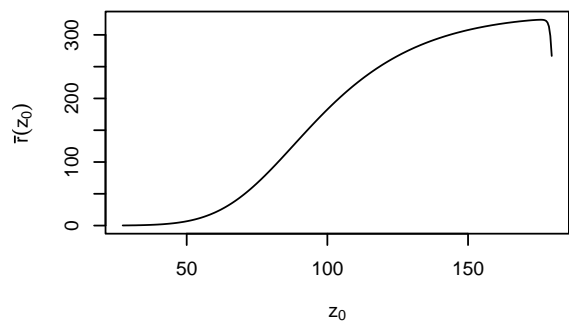
    plot(r_bars[[i]], x = x_seqs[[i]], type = 'l', main = names(r_bars)[i],
         ylab = expression(bar(r)(z[0])),
         xlab = expression(z[0]))

}

```





dddd29**dddd30****dddd33****dddd35****dddd36****dddd37****dddd39**

Troubleshooting and Modifying IPMs

The plot of the output above provides a cautionary tale: $\bar{r}(z_0)$ is negative in model `aaa341`! This is not biologically possible - we can not make negative recruits. We know there is some quirk in that model that produces these results. PADRINO would flag a model with negative numbers in the F kernel when it's built, so we need to check the N kernel for that model for negative values.

```
range(N_kernels$aaa341)
```

```
## [1] -5008.038563      1.020386
```

We have found the problem in the N kernel, but what causes that? It turns out that the P kernel, when discretized, is either close to or exactly singular, and so the determinant $(I - P)$ is very close to singular as well (and negative!). This can happen when the survival function is exactly 1 for some range of initial trait values (Ellner, Childs & Rees 2016, Chapter 3).

```
P <- new_ipms$aaa341$sub_kernels$P
I <- make_i(P)
det(I - P) # negative and very close to 0
```

```
## [1] -1.29379e-09
```

It is important to remember that PADRINO provides models *as they are published*, and does not try to correct these problems. Thus, data in here can cause problems if not treated with care!

We can try to modify the model very slightly to see if we can make this kernel non-singular. We will try to set a [parallel minimum](#) for the s function value in that model, which will hopefully pull our $\det(I - P)$ into positive territory. We need to work with the `proto_ipm` object for this, because we want to propagate the function value changes to the sub-kernels (*i.e.* the P kernel).

We can alter the s function with `vital_rate_exprs<-` and `pdb_new_fun_form()`. We will update it to take the parallel minimum of the published s function, and the maximum value we'd want the function to ideally have (in this case, 0.98).

```
# First, peak at the current functional form
vital_rate_exprs(simple_det_list)$aaa341
```

```
## s: 1/(1 + exp(-(si + ss1 * size_1 + ss2 * size_1^2)))
## g_mean: gi + gs * size_1
## g: stats::dnorm(size_2, g_mean, g_sd)
## Fp: 1/(1 + exp(-(fpi + fps * size_1)))
## Fs: exp(fi + fs * size_1)
## Fd: stats::dnorm(size_2, fd_mean, fd_sd)
```

```
# Now, update it with our desired maximum value
vital_rate_exprs(simple_det_list) <- pdb_new_fun_form(
  list(
    aaa341 = list(
      s = pmin(0.98, 1/(1 + exp(-(si + ss1 * size_1 + ss2 * size_1^2)))
    )
  )
)
```

```
# we will skip rebuilding the whole list - we just want to make sure we have fixed
# this particular model. The species in this model is Lonicera maackii.
```

```
new_lonicera <- pdb_make_ipm(simple_det_list["aaa341"])
```

```
P <- new_lonicera$aaa341$sub_kernels$P
```



```

I <- make_i(P)

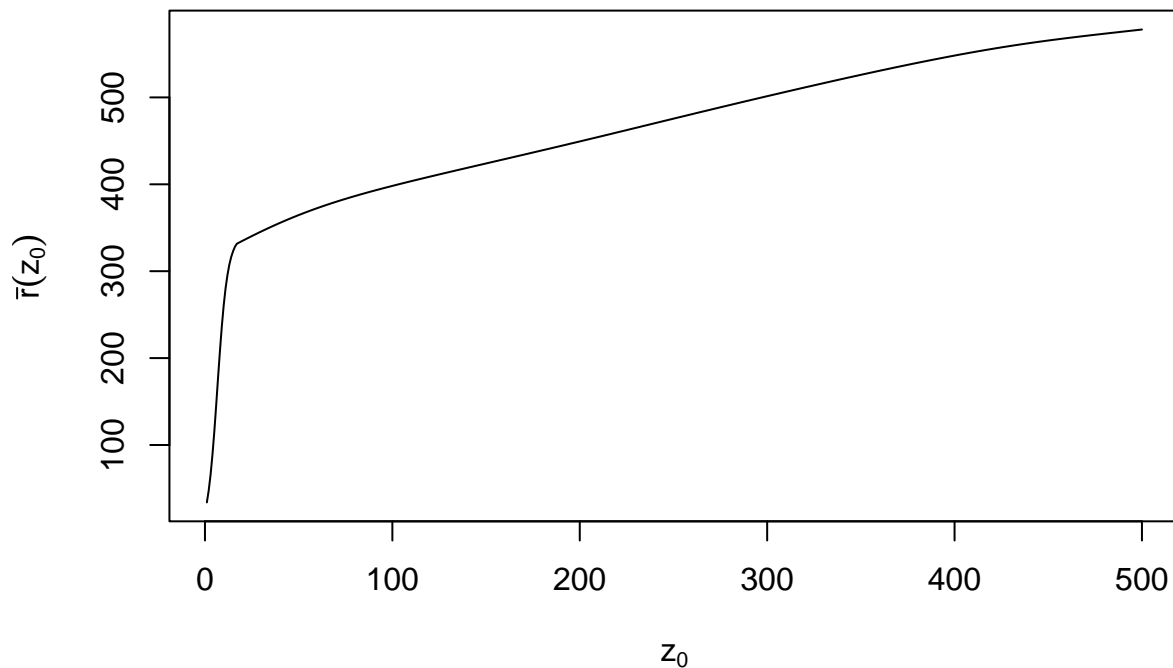
cat("New det(I - P) is: ", det(I - P))

## New det(I - P) is: 1.22853e-05
N <- solve(I - P)
F <- new_lonicera$aaa341$sub_kernels$F

r_bar_lonicera <- as.vector(colSums(F) %*% N)

plot(r_bar_lonicera,
     type = 'l',
     ylab = expression(bar(r)(z[0])),
     xlab = expression(z[0]))

```



```

# Since we are now happier with how this IPM is behaving, we will insert it into
# our list for subsequent analyses
new_ipms$aaa341 <- new_lonicera$aaa341

```

That looks a bit closer to reality!

Vital rate and parameter level analyses

We can also run analyses at the parameter and the vital rate level for PADRINO. These require more care - it is strongly recommended to check the original publications to for the meaning of each parameter and

vital rate. There is simply too much variability in the way vital rates and parameters are estimated in the literature to provide systematic descriptions of them in PADRINO. With this caveat in mind, we will proceed to a couple examples of using vital rate functions and parameters in further analyses.

The ability to perturb function values and parameter estimates is one of the great strengths of IPMs. Furthermore, computing many life history traits requires the values of vital rate functions. Therefore, we took great pains when designing the database to ensure these analyses were still possible. As noted above, they require some additional effort, but are usually worth it. We will step through the code pieces required to extract these below. We will start with an example computing the sensitivity of λ to vital rate function values. After that, we will show how to compute the mean size at death conditional on initial state z_0 and the size at death kernel $\Omega(z', z_0)$, both of which rely on extracting the survival functions.

Vital rate function value perturbations

These require modifying the general sensitivity formula to compute the partial derivative of λ with respect to change in $f(z)$. These expressions depend on the form of the kernel, and so no general formula exists for function value perturbations. However, we can use the chain rule and the general formula for sensitivity to a given perturbation to work it out. The latter formula is:

$$1. \quad \left. \frac{\partial \lambda}{\partial \epsilon} \right|_{\epsilon=0} = \frac{\langle v, Cw \rangle}{\langle v, w \rangle}.$$

Here, v and w are the left and right eigenvectors of the iteration kernel (provided by `left_ev` and `right_ev`), and C is the perturbation kernel, which we will need to identify. We will take the following steps:

1. Identify models we want to use.
2. Inspect the kernel formulae and vital rate functions using `print` methods.
3. Write down the perturbation kernels for each model.
4. Construct the IPM objects and extract v and w .
5. Implement the perturbations in R .

Identifying models (1)

In order to keep things simple, we will work with models where survival only occurs in 1 kernel. There are numerous examples of how to extend these analyses elsewhere (*e.g.* Ellner, Childs & Rees 2016). We can look into this using the `pdb$VitalRateExpr$kernel_id` column in conjunction with the `pdb$VitalRateExpr$demographic_parameter` column.

```
# Find all rows in VitalRateExpr corresponding to survival
init_ind <-
  my_pdb$VitalRateExpr[
    my_pdb$VitalRateExpr$demographic_parameter == "Survival", ]

# Next, select ipm_id's that have survival functions that only show up
# in "P"

keep_ind <- init_ind$ipm_id[init_ind$kernel_id == "P"] %>%
  unique()

# To keep things quick, we will just use the first 3 in this index

keep_ind <- keep_ind[1:3]
```

```
vr_sens_pdb <- pdb_subset(my_pdb, keep_ind)
```

Inspect the kernels and vital rate expressions (2)

Next, we will construct `proto_ipm` objects for each model and check to see how the kernels are constructed:

```
proto_list <- pdb_make_proto_ipm(vr_sens_pdb)
```

```
proto_list[[1]]
```

```
## A simple, density independent, deterministic proto_ipm with 2 kernels defined:
```

```
## P, F
```

```
##
```

```
## Kernel formulae:
```

```
##
```

```
## P: s * g
```

```
## F: r * fn * pE * d
```

```
##
```

```
## Vital rates:
```

```
##
```

```
## s: 1/(1 + exp(-(s_b + s_m * lnsizesize_1)))
```

```
## g_mean: g_b + g_m * lnsizesize_1
```

```
## g_var: sqrt(gv_b + gv_m * lnsizesize_1)
```

```
## g: stats::dnorm(lnsizesize_2, g_mean, g_var)
```

```
## r: 1/(1 + exp(-(r_b + r_m * lnsizesize_1)))
```

```
## fn: exp(fn_b + fn_m * lnsizesize_1)
```

```
## d: stats::dexp(lnsizesize_2, 1/d_mean)
```

```
##
```

```
## Parameter names:
```

```
##
```

```
## [1] "s_b"      "s_m"      "g_b"      "g_m"      "gv_b"     "gv_m"     "r_b"      "r_m"      "fn_b"     "fn_m"     "t_r"
```

```
## [13] "pE"
```

```
##
```

```
## All parameters in vital rate expressions found in 'data_list': TRUE
```

```
##
```

```
## Domains for state variables:
```

```
##
```

```
## lnsizesize: lower_bound = 0, upper_bound = 5, n_meshpoints = 500
```

```
##
```

```
## Population states defined:
```

```
##
```

```
## n_lnsizesize: Pre-defined population state.
```

```
##
```

```
## Internally generated model iteration procedure:
```

```
##
```

```
## n_lnsizesize_t_1: right_mult(kernel = P, vectr = n_lnsizesize_t) + right_mult(kernel = F,
```

```
## vectr = n_lnsizesize_t)
```

```
proto_list[[2]]
```

```
## A simple, density independent, deterministic proto_ipm with 2 kernels defined:
```

```
## P, F
```

```
##
```

```
## Kernel formulae:
```

```

##
## P: s * g
## F: r * fn * pE * d
##
## Vital rates:
##
## s: 1/(1 + exp(-(s_b + s_m * lnsizes_1)))
## g_mean: g_b + g_m * lnsizes_1
## g_var: sqrt(gv_b + gv_m * lnsizes_1)
## g: stats::dnorm(lnsizes_2, g_mean, g_var)
## r: 1/(1 + exp(-(r_b + r_m * lnsizes_1)))
## fn: exp(fn_b + fn_m * lnsizes_1)
## d: stats::dexp(lnsizes_2, 1/d_mean)
##
## Parameter names:
##
## [1] "s_b"      "s_m"      "g_b"      "g_m"      "gv_b"     "gv_m"     "r_b"      "r_m"      "fn_b"     "fn_m"     "t_r"
## [13] "pE"
##
## All parameters in vital rate expressions found in 'data_list': TRUE
##
## Domains for state variables:
##
## lnsizes: lower_bound = 0, upper_bound = 5, n_meshpoints = 500
##
## Population states defined:
##
## n_lnsizes: Pre-defined population state.
##
## Internally generated model iteration procedure:
##
## n_lnsizes_t_1: right_mult(kernel = P, vectr = n_lnsizes_t) + right_mult(kernel = F,
##      vectr = n_lnsizes_t)
proto_list[[3]]

## A simple, density independent, deterministic proto_ipm with 2 kernels defined:
## P, F
##
## Kernel formulae:
##
## P: s * g
## F: rep_p * es_p * sdl_s * n_infl * n_fl * n_seed
##
## Vital rates:
##
## s: ssurv * wsurv
## ssurv: exp(ssurv_i + ssurv_s * ((log(size_1^3) - smlv)/sslv) + ssurv_el *
##      elev_s)/(1 + exp(ssurv_i + ssurv_s * ((log(size_1^3) - smlv)/sslv) +
##      ssurv_el * elev_s))
## wsurv: exp(wsurv_i + wsurv_s * ((log(size_1^3) - wmlv)/wslv) + wsurv_f *
##      cumfrost)/(1 + exp(wsurv_i + wsurv_s * ((log(size_1^3) -
##      wmlv)/wslv) + wsurv_f * cumfrost))
## g_mean: sign(growth) * abs(growth)^(1/3)
## growth: g_i + g_el * elev_g + g_frost * annfrost + g_s * (size_1^3)

```

```

## g: stats::dnorm(size_2, g_mean, g_sd)
## rep_p: fl_p * germ_p
## fl_p: exp(fl_i + fl_s * ((log(size_1^3) - fmlv)/fslv))/(1 + exp(fl_i +
##     fl_s * ((log(size_1^3) - fmlv)/fslv)))
## germ_p: exp(germ_i + germ_el * elev_germ)/(1 + exp(germ_i + germ_el *
##     elev_germ))
## n_infl: exp(infl_n)
## n_fl: exp(fl_n)
## n_seed: exp(seed_i)
## sdl_s: stats::dnorm(size_2, sdl_mean, sdl_sd)
##
## Parameter names:
##
## [1] "ssurv_i"    "ssurv_el"   "ssurv_s"    "wsurv_i"    "wsurv_s"    "wsurv_f"    "g_i"        "g_el"
## [10] "g_s"        "g_sd"       "fl_i"       "fl_s"       "germ_i"     "germ_el"    "es_p"       "sdl_mean"
## [19] "infl_n"     "fl_n"       "seed_i"     "smlv"       "sslv"       "wmlv"       "wslv"       "fmlv"
## [28] "annfrost"   "cumfrost"   "elev_germ"  "elev_g"     "elev_s"
##
## All parameters in vital rate expressions found in 'data_list': TRUE
##
## Domains for state variables:
##
## size: lower_bound = 2, upper_bound = 55, n_meshpoints = 1000
##
## Population states defined:
##
## n_size: Pre-defined population state.
##
## Internally generated model iteration procedure:
##
## n_size_t_1: right_mult(kernel = P, vectr = n_size_t) + right_mult(kernel = F,
##     vectr = n_size_t)

```

We can see that for each IPM, $P = s(z) * G(z', z)$. In the third models, $s(z)$ is comprised of two additional functions, which we can perturb individually. We will show a quick example of that after applying our perturbation kernels to all 3.

Write out the perturbation kernels (3)

Our perturbation kernel for all 3 models will take the following form: $C(z', z) = \delta_{z_0}(z)G(z', z)$. With a little rearranging (see Ellner, Childs, & Rees 2016 Chapter 4), we find the following:

$$\frac{\partial \lambda}{\partial s(z_0)} = \frac{\int v(z')G(z', z_0)w(z_0)dz'}{\int v(z)w(z)dz} = \frac{(vG) \circ w}{\langle v, w \rangle}.$$

The second portion of the equations above is the first part re-written to use operator notation (which drops the z s and z' s for brevity). The \circ denotes point-wise multiplication.

Implement the models (4)

Now that we have written down our perturbation formulae, we need to rebuild the models, and make use of some non-standard arguments to `pdb_make_ipm`. By default, *Rpadrino* does not return the vital rate functions values. To get those, we need to specify `return_all_envs = TRUE` in the `addl_args` list.

```
arg_list <- lapply(keep_ind, function(x) list(return_all_envs = TRUE)) %>%
  setNames(keep_ind)

ipm_list <- pdb_make_ipm(proto_list, addl_args = arg_list)

r_evs <- right_ev(ipm_list)
l_evs <- left_ev(ipm_list)
```

Implement the perturbations (5)

Next, we need to implement the formula above. This is fairly straightforward, and we will make use of another function in *Rpadrino*: `vital_rate_funs()` (not to be confused with `vital_rate_exprs()`!). This extracts the vital rate function values from each model and returns them in a named list (`vital_rate_exprs()` extracts the expressions that create these values). Let's see what this looks like:

```
vr_funs <- vital_rate_funs(ipm_list)
```

```
vr_funs$aaaa34
```

```
## $P
```

```
## s (not yet discretized): A 500 x 500 kernel with minimum value: 0.3638 and maximum value: 0.852
```

```
## g_mean (not yet discretized): A 500 x 500 kernel with minimum value: 1.1114 and maximum value: 3.680
```

```
## g_var (not yet discretized): A 500 x 500 kernel with minimum value: 0.9749 and maximum value: 0.9749
```

```
## g (not yet discretized): A 500 x 500 kernel with minimum value: 0 and maximum value: 0.1293
```

```
##
```

```
## $F
```

```
## r (not yet discretized): A 500 x 500 kernel with minimum value: 0.0062 and maximum value: 0.9969
```

```
## fn (not yet discretized): A 500 x 500 kernel with minimum value: 43.05 and maximum value: 1280.9888
```

```
## d (not yet discretized): A 500 x 500 kernel with minimum value: 0 and maximum value: 0.1481
```

```
vr_funs$aaa144
```

```
## $P
```

```
## s (not yet discretized): A 1000 x 1000 kernel with minimum value: 0.9712 and maximum value: 0.9997
```

```
## ssurv (not yet discretized): A 1000 x 1000 kernel with minimum value: 0.9729 and maximum value: 0.999
```

```
## wsurv (not yet discretized): A 1000 x 1000 kernel with minimum value: 0.9983 and maximum value: 1
```

```
## g_mean (not yet discretized): A 1000 x 1000 kernel with minimum value: 18.6951 and maximum value: 42
```

```
## growth (not yet discretized): A 1000 x 1000 kernel with minimum value: 6534.029 and maximum value: 7
```

```
## g (not yet discretized): A 1000 x 1000 kernel with minimum value: 0 and maximum value: 0.0798
```

```
##
```

```
## $F
```

```
## rep_p (not yet discretized): A 1000 x 1000 kernel with minimum value: 0 and maximum value: 0.0049
```

```
## fl_p (not yet discretized): A 1000 x 1000 kernel with minimum value: 0 and maximum value: 0.3948
```

```
## germ_p (not yet discretized): A 1000 x 1000 kernel with minimum value: 0.0124 and maximum value: 0.0
```

```
## n_infl (not yet discretized): A 1000 x 1000 kernel with minimum value: 2.4843 and maximum value: 2.4
```

```
## n_fl (not yet discretized): A 1000 x 1000 kernel with minimum value: 131.6307 and maximum value: 131
```

```
## n_seed (not yet discretized): A 1000 x 1000 kernel with minimum value: 13.1971 and maximum value: 13
```

```
## sdl_s (not yet discretized): A 1000 x 1000 kernel with minimum value: 0 and maximum value: 0.3988
```

We see from the printed values that each vital rate function contains the complete $n \times n$ set of values for each combination of meshpoints. Additionally, it warns us that these are not yet integrated. This is actually a good thing - we want the continuous function values for the sensitivity, not the discretized values. We know from our formula above that we need to extract $G(z', z)$, and that these are named `g` in each model. This will not always be true for PADRINO, so care must be taken at this step to make sure you extract the correct values!

Recall that we are about to implement: $\frac{(vG) \circ w}{\langle v, w \rangle}$. The $\langle \dots \rangle$ is the inner product of v, w , and so we also need the value of dz to implement the denominator. We will get those using `int_mesh()` again.

```
mesh <- lapply(ipm_list, function(x) int_mesh(x, full_mesh = FALSE))
d_zs <- lapply(mesh, function(x) x[[1]])

sens_list <- lapply(seq_along(vr_funs),
  function(idx, r_evs, vr_funs, l_evs, d_zs) {

    # Extract objects to temporary values so the formula is more
    # readable

    G <- vr_funs[[idx]]$P$g
    v <- unlist(l_evs[[idx]])
    w <- unlist(r_evs[[idx]])
    d_z <- d_zs[[idx]]

    numerator <- as.vector((v %*% G) * w)
    denominator <- sum(v * w * d_z)

    numerator / denominator

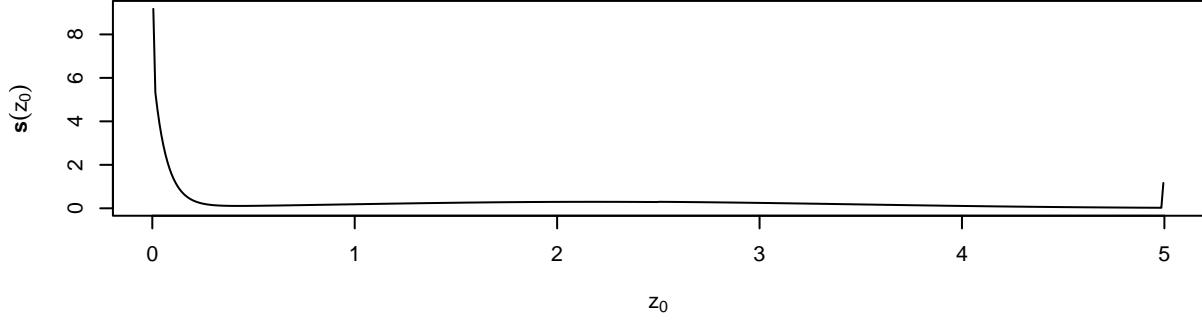
  },
  r_evs = r_evs,
  l_evs = l_evs,
  vr_funs = vr_funs,
  d_zs = d_zs)

par(mfrow = c(3, 1))

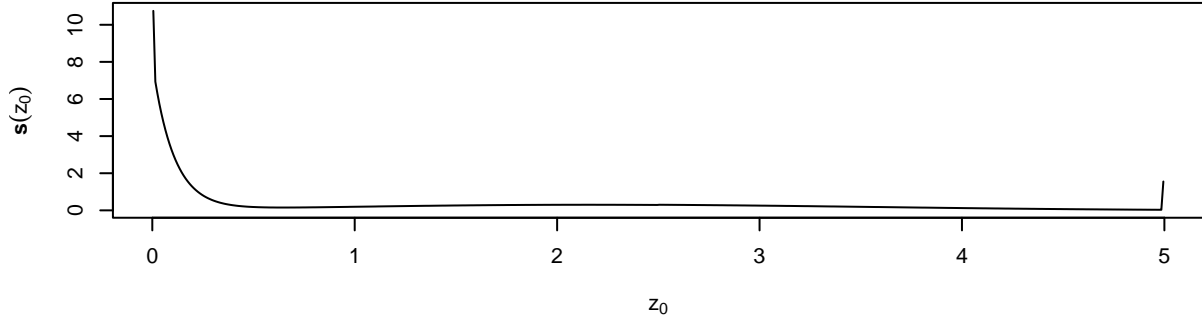
for(i in seq_along(sens_list)) {

  plot(sens_list[[i]], x = mesh[[i]][[2]],
    type = "l",
    main = names(mesh)[i],
    ylab = expression(bold(s)(z[0])),
    xlab = expression(z[0]))
}
```

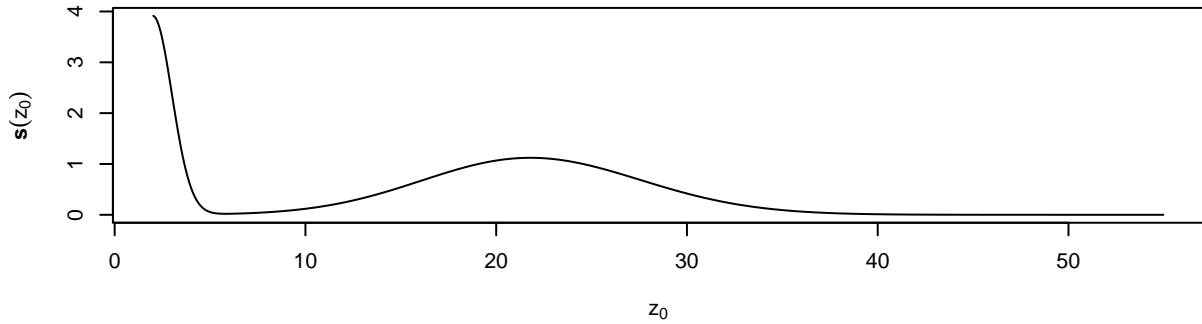
aaaa34



aaaa36



aaa144



Other perturbation kernels

As mentioned above, the survival function in the last IPM is comprised of two additional functions: **ssurv** and **wsurv**. Thus, we could re-write the P kernel as $P(z', z) = s_s(z) * s_w(z) * G(z', z)$. Thus, if we wanted to know the effect of perturbing only $s_w(z)$, we would re-write our perturbation kernel as $C(z', z) = \delta(z_0) * s_s(z) * G(z', z)$, and our perturbation formula (in operator notation) becomes $\mathbf{s}_s(z_0) = \frac{(vs_s G) \circ w}{\langle v, w \rangle}$. We will drop the first model from our lists because this analysis does not apply to it. We can implement this by slightly modifying the code above:


```

s_s <- vr_funs[[3]]$P$ssurv
G <- vr_funs[[3]]$P$g
v <- unlist(l_evs[[3]])
w <- unlist(r_evs[[3]])
d_z <- d_zs[[3]]

numerator <- as.vector((v %*% (s_s * G)) * w)
denominator <- sum(v * w * d_z)

sens_s_w <- numerator / denominator

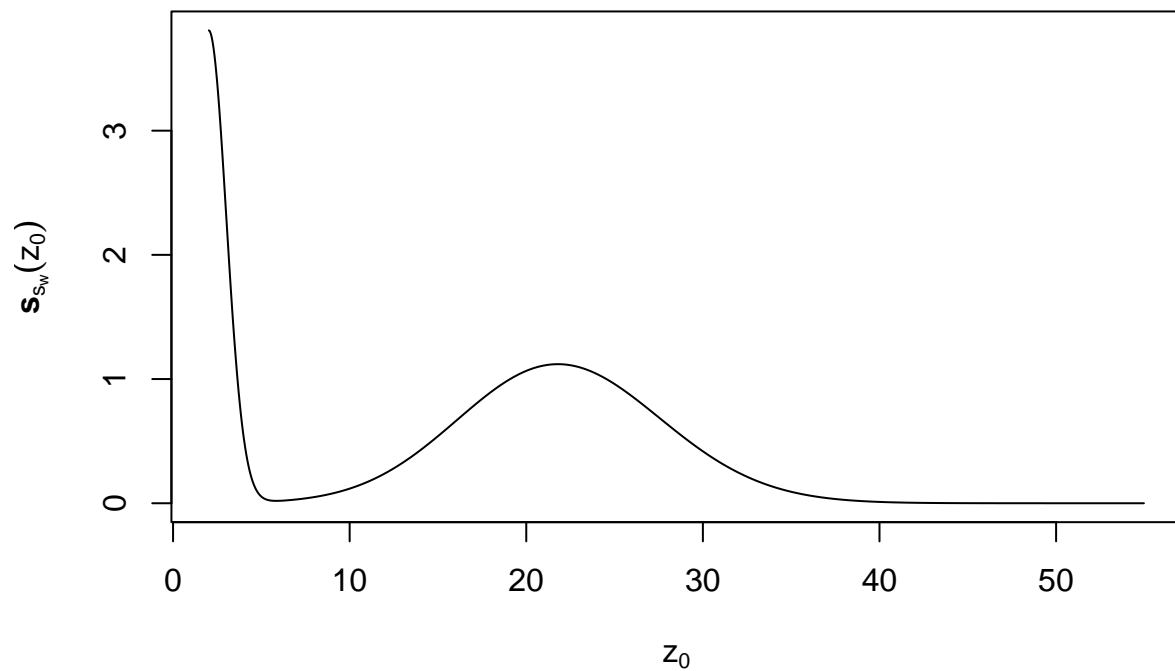
par(mfrow = c(1, 1))

mesh_ps <- mesh[[3]][[2]]
nm <- names(mesh)[3]

plot(y = sens_s_w, x = mesh_ps,
     type = "l",
     main = nm,
     ylab = expression(bold(s)[s[w]](z[0])),
     xlab = expression(z[0]))

```

aaa144



Mean size at death and size at death kernels

We can also use vital rate function values in conjunction with sub-kernels to implement calculations of life history traits. For this example, we will examine mean size at death and the size at death kernel. These are given by the following equations:

Mean size at death = $\bar{\omega} = (\mathbf{i} \circ (1 - s))N$ and size at death kernel = $\Omega(z', z_0) = (1 - s(z')) * N(z', z_0)$.

In these equations, s and $s(z')$ are survival functions, and N is the fundamental operator, which is defined as $N = (I - P)^{-1}$, where P is the survival/growth kernel from the IPM and I is an identity kernel (analogous to an identity matrix). Fortunately, we can reuse our `N_kerns` code from above. However, it is important to remember that the $(1 - s)$ term represents all mortality pathways, and species may have more than one way to die (*e.g.* monocarpic perennials die through natural mortality as well as the flowering process). Therefore, we also want to check our kernel formulae and see if those include additional terms that may represent alternative mortality pathways. Additionally, we need to get the meshpoints which correspond to \mathbf{i} .

This time, we will use considerably more IPMs - we will get those from the `new_ipms` object we created earlier. However, we need to rebuild them with `return_all_envs = TRUE` so that we can access the vital rate function values.

```
arg_list <- lapply(names(new_ipms), function(x) list(return_all_envs = TRUE,
                                                    iterations      = 250)) %>%
  setNames(names(new_ipms))

new_ipms <- pdb_make_ipm(simple_det_list, addl_args = arg_list)

# Removing our these problem IPMs we identified before

keep_ind <- setdiff(names(new_ipms), c("dddd24", "dddd26",
                                       "dddd40", "dddd41"))

new_ipms <- new_ipms[keep_ind]

kernel_formulae(new_ipms)

## $aaaa34
## P: s * g
## F: r * fn * pE * d
## $aaaa36
## P: s * g
## F: r * fn * pE * d
## $aaa144
## P: s * g
## F: rep_p * es_p * sdl_s * n_infl * n_fl * n_seed
## $aaa227
## P: s * g
## F: rep_p * es_p * sdl_s * n_infl * n_fl * n_seed
## $aaa310
## P: s * g
## F: f_n * f_d
## $aaa323
## P: s * g
## F: p_es * sdl_size * n_seeds
## $aaa326
## P: (1 - p_fl) * s * g
```

```

## F: p_fl * fec1 * sdl_size * est_p
## $aaa341
## P: s * g
## F: Ep * Fp * Fs * Fd
## $aaa351
## P: s * g
## F: Pf * Nfruit * Nseeds * Pe * Fd
## $aaa385
## P: s * g
## F: f * fd
## $cccc1
## P: s * g
## F: p_r * r_s * r_d * r_r
## $dddd3
## P: s * g
## F: d * r
## $dddd4
## P: s * g
## F: d * r
## $dddd5
## P: s * g
## F: d * r
## $dddd7
## P: s * g
## F: d * r
## $dddd10
## P: s * g * d_len
## F: d * r
## $ddd29
## P: s * g
## F: (s * r * pHS * d)/2
## $ddd30
## P: s * g
## F: s * r * pg * 0.5 * d
## $ddd33
## P: s * g
## F: rp * f * EP * d
## $ddd35
## P: s * g
## F: p_fertile * polyyps * fec * p_est * d
## $ddd36
## P: s * g
## F: p_fertile * polyyps * fec * p_est * d
## $ddd37
## P: s * g
## F: d * r
## $ddd39
## P: s * g
## F: f1 * f2 * f3 * f4 * f5 * d * 0.5

```

Upon further inspection, model ID aaa326 contains a $(1 - p_{fl})$ term, which represents mortality due to flowering. We need to include this in the calculations above. We do this like so:

$$\bar{\omega} = (\mathbf{i} * (p_{fl} + (1 - p_{fl}) * (1 - s)))N,$$

and

$$\Omega(z', z_0) = (p_{fl}(z') + (1 - p_{fl}(z')) * (1 - s(z'))N(z', z_0).$$

This can be summarized by saying “a plant dies with flowering probability p_{fl} , and if the plant does not flower ($1 - p_{fl}$), it dies with probability $1 - s$.” We are now ready to proceed with our calculations!

```
N_kerns <- lapply(new_ipms, function(x) {

  P <- x$sub_kernels$P
  I <- make_i(P)

  solve(I - P)

})

surv_funs <- lapply(new_ipms, function(x) {
  vital_rate_funs(x)$P$s
})

p_flower <- vital_rate_funs(new_ipms$aaa326)$P$p_fl
```

The `i` corresponds to the sizes in each IPM. Thus, we want to get the meshpoints, which we do with `int_mesh` like before. The `lapply(x[[2]])` extracts the value of `z_1`, as this is always the second entry in the list of 3 returned by `int_mesh` (and we do not need the `d_z` or `z_2` values).

```
i_vals <- int_mesh(new_ipms, full_mesh = FALSE) %>%
  lapply(function(x) x[[2]])
```

Now, we just have to implement the calculations:

```
omega_bar_z <- lapply(seq_along(new_ipms),
  function(index, i_vals, surv_funs, N_kerns, p_flower) {

    id <- names(i_vals)[index]
    i <- i_vals[[index]]

    # The survival function is represented as a bivariate
    # function in ipmr even though it is actually a univariate
    # function of z. Thus, we need to pull out the
    # univariate form of it to ensure we get the correct
    # result from (1 - s) (_i.e._ a vector, not an array!).
    # The correct univariate form is given by the rows,
    # as every column contains the same values.

    s <- surv_funs[[index]][1, ]
    N <- N_kerns[[index]]

    if(id == "aaa326") {
      # Same indexing as the survival function
      p_fl <- p_flower[1, ]

      out <- (i * (p_fl + (1-p_fl) * (1 - s))) %*% N
    } else {

      out <- (i * (1 - s)) %*% N
```

```

    }

    return(out)
  },
  i_vals = i_vals,
  surv_funs = surv_funs,
  N_kerns = N_kerns,
  p_flower = p_flower) %>%
setNames(names(i_vals))

Omega_z0_z <- lapply(seq_along(new_ipms),
  function(index, surv_funs, N_kerns, p_flower) {
    id <- names(surv_funs)[index]

    s <- surv_funs[[index]][1, ]
    N <- N_kerns[[index]]

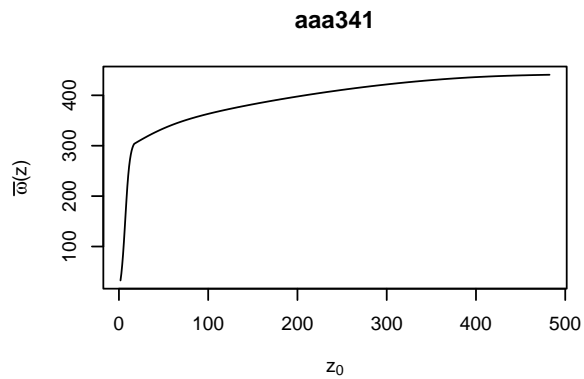
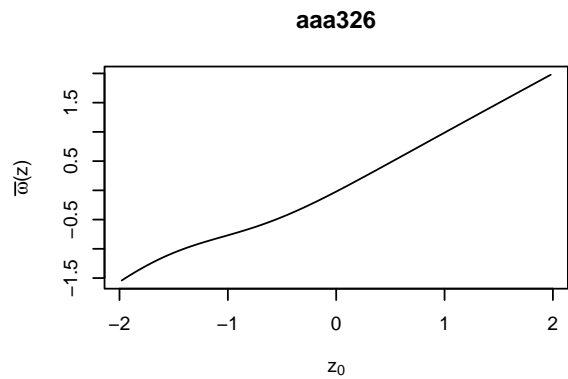
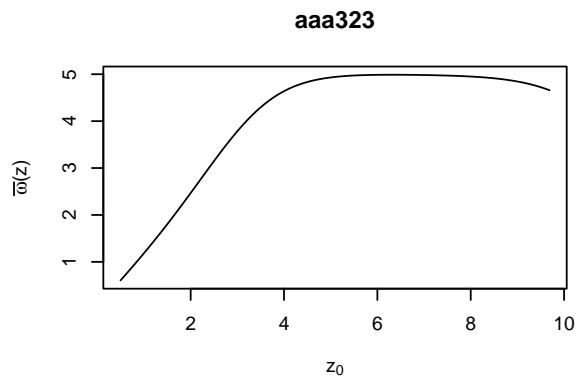
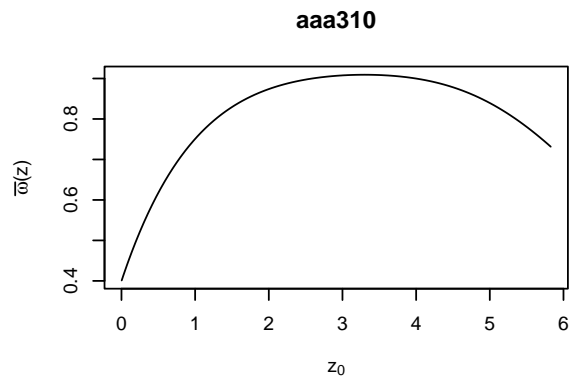
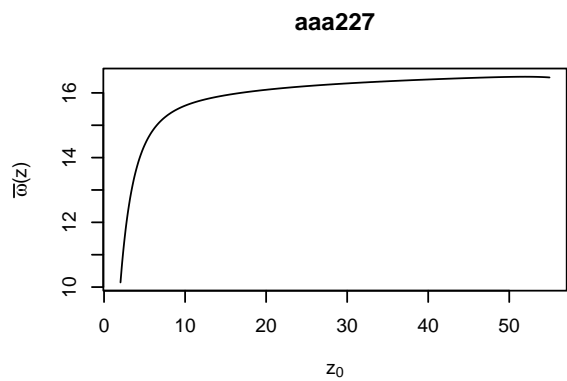
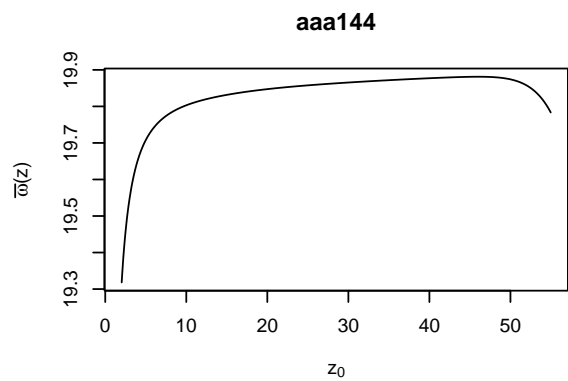
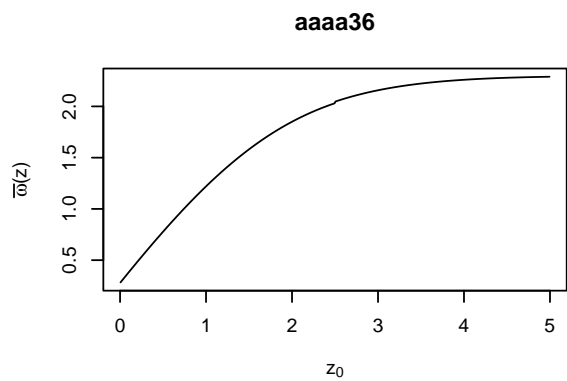
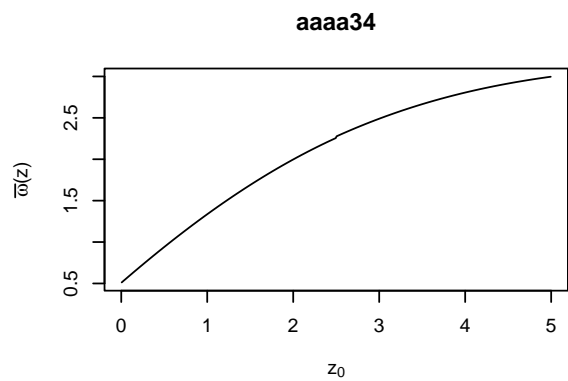
    if(id == "aaa326") {
      p_fl <- p_flower
      out <- (p_fl + (1-p_fl) * (1 - s)) %**% N
    } else {
      out <- (1 - s) * N
    }

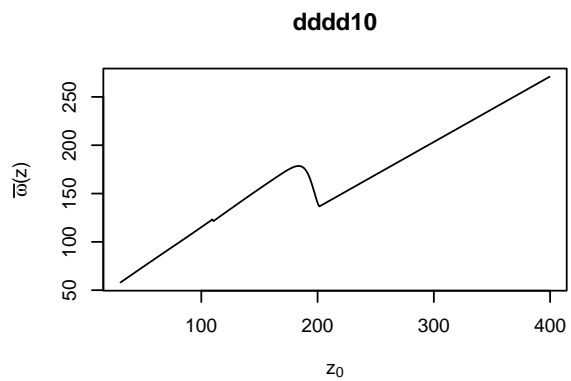
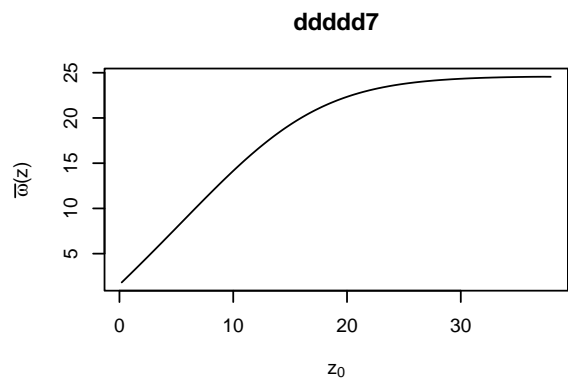
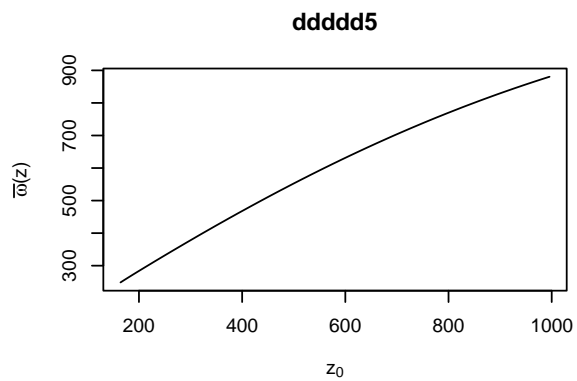
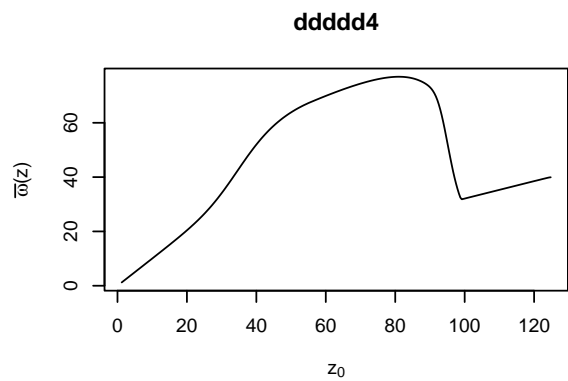
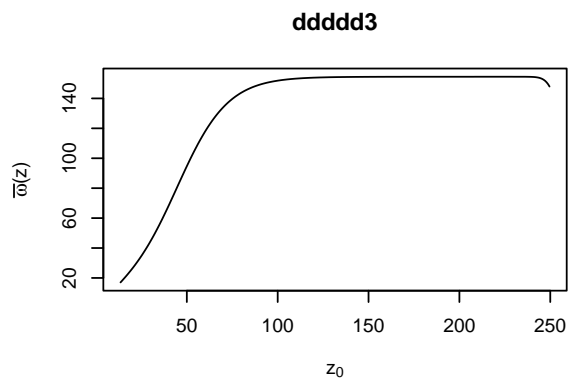
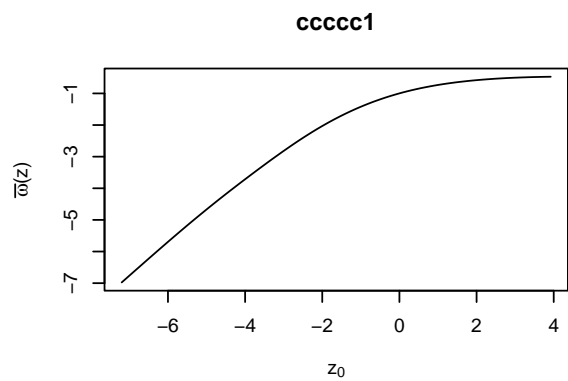
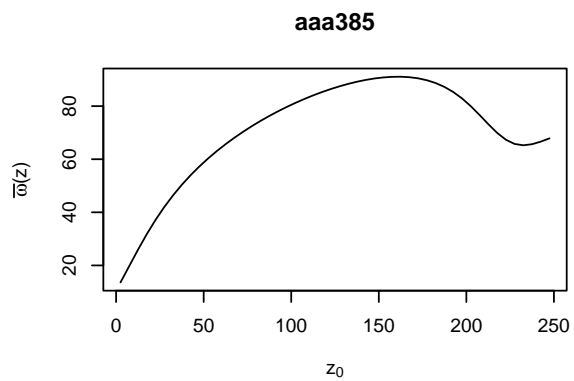
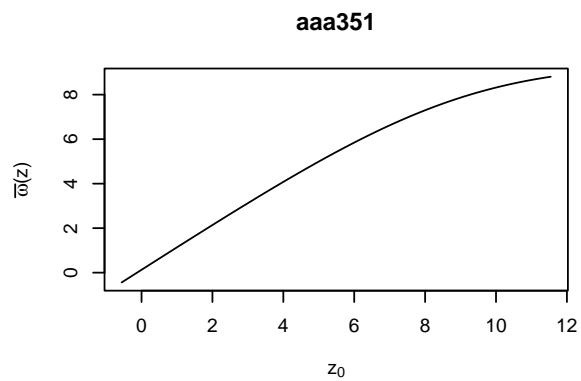
    return(out)
  },
  surv_funs = surv_funs,
  N_kerns = N_kerns,
  p_flower = p_flower) %>%
setNames(names(i_vals))

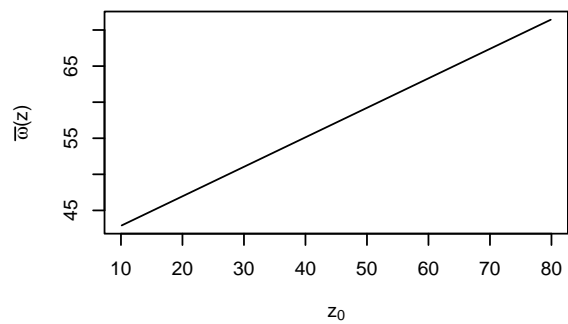
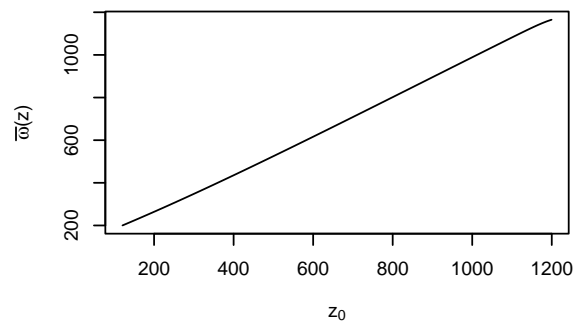
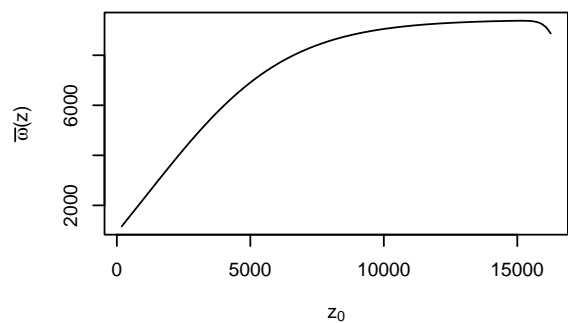
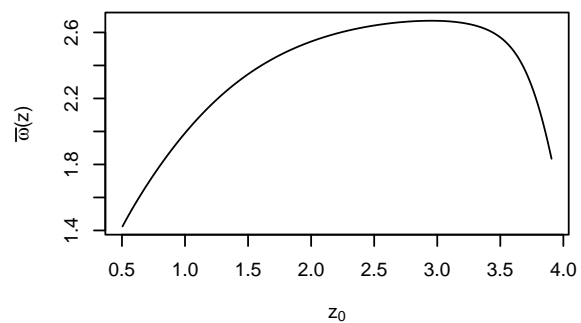
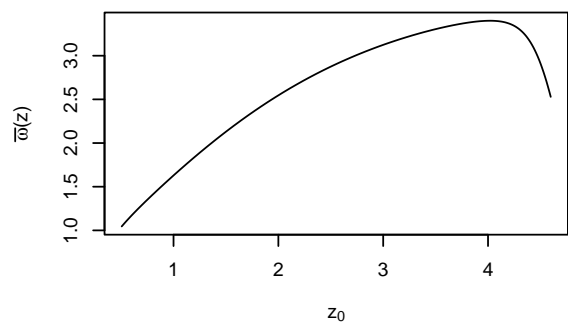
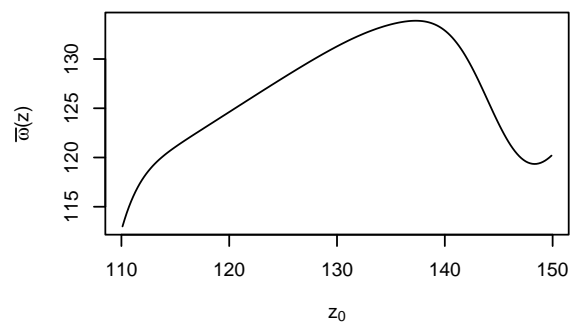
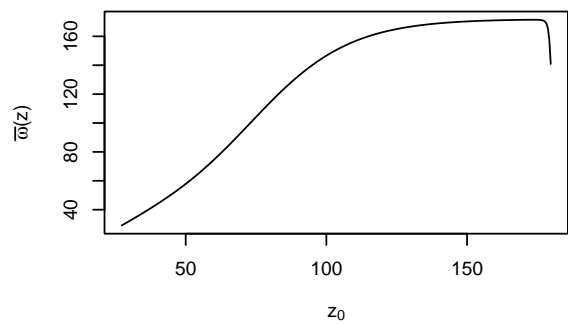
par(mfrow = c(4, 2))

for(i in seq_along(omega_bar_z)) {
  plot(x = i_vals[[i]],
       y = as.vector(omega_bar_z[[i]]),
       type = "l",
       main = names(i_vals)[i],
       xlab = expression(paste(z[0])),
       ylab = expression(paste(bar(omega), "(z)")))
}

```





dddd29**dddd30****dddd33****dddd35****dddd36****dddd37****dddd39**

This is an interesting variety of relationships! Sometimes, it increases linearly, whereas other times we get a parabolic relationships. This is a good first pass on an analysis, though subsequent digging would likely reveal quirks in some of these models that we'd need to address more thoroughly. After these exercises, you should have the tools to do just that!

Recap

We first showed how to subset PADRINO using the Metadata table, as well as a few others. Next, we demonstrated how to rebuild kernels from the database, as well as basic analyses such as deterministic population growth rates and perturbations. Next we moved into some examples of life cycle properties, such as recruit production and size at death. Along the way, we encountered some issues with data that required us to manipulate the underlying `proto_ipms`. This is far from an exhaustive display of potential analyses. However, this case study should serve as a guide for posing questions and solving issues with PADRINO.

Appendix 5: PADRINO Case Study 2

Overview

In this case study, we demonstrate how PADRINO can be used in conjunction with (1) your own unpublished data, and (2) external data repositories.

In many cases, we may wish to combine data that we’ve collected and not yet published with data from PADRINO. For example, we may wish to compare the vital rates or sensitivities of our (as-yet-unpublished) study system with those from published IPMs on e.g. similar species. We describe how to do this in Section 1 of this case study.

Using PADRINO in conjunction with external data sources presents unique opportunities to address synthetic questions in ecology, evolutionary biology, and conservation. External data sources could (though are not limited to) include climate data, species range distributions, phylogenies, or life tables. We describe how to use PADRINO in conjunction with BIEN in Section 2 of this case study.

Combining your own data with PADRINO

In this section, we demonstrate how to combine data from PADRINO with a users own unpublished data. The first part of the code generates two IPMs - these are our “unpublished IPMs”. The next section shows how to combine our unpublished IPMs with those stored in PADRINO, and how to perform simple analyses (i.e. calculate population growth rate) across the combined set of IPMs.

Creating our own IPMs

The goal of this case study is to show how to combine data, and not necessarily how to use `ipmr`. `ipmr` is extensively documented on the [project’s website](#) and in the [publication describing the package](#). Therefore, the next few chunks of code assume you have already consulted these resources and will have a reasonable understanding of what’s going on. If you have not already consulted these, please do so now.

Our first “homemade” IPM will be a general IPM. For now, we are only going to construct `proto_ipm` objects for each one of these homemade IPMs. Once we have our PADRINO IPMs selected, we will splice everything together and generate actual IPM objects.

```
# Loading Rpadrino automatically loads ipmr, so we do not need to load both.  
library(Rpadrino)
```

```
# Set up the initial population conditions and parameters.  
# These are hypothetical values and do not correspond to any particular  
# species.
```

```
data_list <- list(  
  g_int      = 5.781,  
  g_slope    = 0.988,  
  g_sd       = 20.55699,  
  s_int      = -0.352,  
  s_slope    = 0.122,  
  s_slope_2  = -0.000213,  
  r_r_int    = -11.46,  
  r_r_slope  = 0.0835,  
  r_s_int    = 2.6204,  
  r_s_slope  = 0.01256,  
  r_d_mu     = 5.6655,
```

```

r_d_sd    = 2.0734,
e_p       = 0.15,
g_i       = 0.5067,
sb_surv   = 0.2
)

# Lower bound, upper bound, and number of meshpoints.
L <- 1.02
U <- 624
n <- 500

# Initialize a population vector. The continuous state will have 500 meshpoints,
# and we will pretend there's a seedbank.

init_pop_vec <- runif(500)
init_seed_bank <- 20

my_general_ipm <- init_ipm(sim_gen = "general", di_dd = "di", det_stoch = "det") %>%
  define_kernel(
    name      = "p",
    formula   = s * g * d_ht,
    family    = "CC",
    g         = dnorm(ht_2, g_mu, g_sd),
    g_mu      = g_int + g_slope * ht_1,
    s         = plogis(s_int + s_slope * ht_1 + s_slope_2 * ht_1^2),
    data_list = data_list,
    states    = list(c('ht')),
    uses_par_sets = FALSE,
    evict_cor = TRUE,
    evict_fun = truncated_distributions('norm',
                                         'g')
  ) %>%
  define_kernel(
    name      = "go_discrete",
    formula   = r_r * r_s * d_ht,
    family    = 'CD',
    r_r       = plogis(r_r_int + r_r_slope * ht_1),
    r_s       = exp(r_s_int + r_s_slope * ht_1),
    data_list = data_list,
    states    = list(c('ht', "b")),
    uses_par_sets = FALSE
  ) %>%
  define_kernel(
    name      = "stay_discrete",
    family    = "DD",
    formula   = sb_surv * (1 - g_i),
    data_list = data_list,
    states    = list(c("b")),
    uses_par_sets = FALSE
  ) %>%
  define_kernel(
    name      = 'leave_discrete',
    formula   = e_p * g_i * r_d * d_ht,

```

```

r_d      = dnorm(ht_2, r_d_mu, r_d_sd),
family   = 'DC',
data_list = data_list,
states   = list(c('ht', "b")),
uses_par_sets = FALSE,
evict_cor = TRUE,
evict_fun = truncated_distributions('norm',
                                   'r_d')
) %>%
define_impl(
  list(
    P      = list(int_rule   = "midpoint",
                  state_start = "ht",
                  state_end   = "ht"),
    go_discrete = list(int_rule   = "midpoint",
                      state_start = "ht",
                      state_end   = "b"),
    leave_discrete = list(int_rule   = "midpoint",
                         state_start = "b",
                         state_end   = "ht"),
    stay_discrete = list(int_rule   = "midpoint",
                        state_start = "b",
                        state_end   = "b")
  )
) %>%
define_domains(
  ht = c(L, U, n)
) %>%
define_pop_state(
  pop_vectors = list(
    n_ht = init_pop_vec,
    n_b  = init_seed_bank
  )
)

```

Our next IPM will be a simple one:

*# Another hypothetical model. These parameters also do not correspond to any
species.*

```

my_data_list = list(s_int   = -2.2,
                    s_slope  = 0.25,
                    g_int    = 0.2,
                    g_slope  = 0.99,
                    sd_g     = 0.7,
                    r_r_int  = 0.003,
                    r_r_slope = 0.015,
                    r_s_int  = 0.45,
                    r_s_slope = 0.075,
                    mu_fd    = 2,
                    sd_fd    = 0.3)

my_simple_ipm <- init_ipm(sim_gen = "simple",
                        di_dd    = "di",

```

```

det_stoch = "det") %>%

define_kernel(
  name      = "P_simple",
  family    = "CC",
  formula    = s * G,
  s          = plogis(s_int + s_slope * dbh_1),
  G          = dnorm(dbh_2, mu_g, sd_g),
  mu_g       = g_int + g_slope * dbh_1,
  data_list = my_data_list,
  states     = list(c('dbh')),
  evict_cor  = TRUE,
  evict_fun  = truncated_distributions(fun      = 'norm',
                                       target    = 'G')
) %>%
define_kernel(
  name      = 'F_simple',
  formula    = r_r * r_s * r_d,
  family     = 'CC',
  r_r        = plogis(r_r_int + r_r_slope * dbh_1),
  r_s        = exp(r_s_int + r_s_slope * dbh_1),
  r_d        = dnorm(dbh_2, mu_fd, sd_fd),
  data_list  = my_data_list,
  states     = list(c('dbh')),
  evict_cor  = TRUE,
  evict_fun  = truncated_distributions(fun      = 'norm',
                                       target    = 'r_d')
) %>%
define_impl(
  make_impl_args_list(
    kernel_names = c("P_simple", "F_simple"),
    int_rule     = rep("midpoint", 2),
    state_start  = rep("dbh", 2),
    state_end    = rep("dbh", 2)
  )
) %>%
define_domains(
  dbh = c(0,
          50,
          100
  )
) %>%
define_pop_state(
  n_dbh = runif(100)
)

my_ipm_list = list(ipm_1 = my_general_ipm, ipm_2 = my_simple_ipm)

```

Combining user-defined and PADRINO-defined IPMs

Next, we will create a list of `proto_ipm` objects from PADRINO, and then put everything together. For simplicity, we will select a small number of plant species. The `pdb` object is contained within the `Rpadrino` package. It is not a complete version of PADRINO. We will use the complete data set in the next section,

accessed with `pdb_download()`.

```
data(pdb)

id_index <- c(
  paste0( "aaaa", c(34, 55)),
  paste0("aaa", c(310, 312, 339, 341, 353, 388))
)

small_db <- pdb_subset(pdb, id_index)
```

Next, we need to create a list that holds both the PADRINO IPMs and the ones we created above. After that, we can call `pdb_make_ipm()` on the combined data set, and voila! We have our database IPMs and our own homemade ones.

```
proto_list <- c(
  pdb_make_proto_ipm(small_db),
  my_ipm_list
)

## 'ipm_id' aaa310 has the following notes that require your attention:
## aaa310: 'Geo and time info retrieved from COMPADRE (v.X.X.X.4)'
## 'ipm_id' aaa388 has the following notes that require your attention:
## aaa388: 'Same data as AAA388. State variable Height (Cm)'
```

Great! In that single step, we combined PADRINO IPMs with our own IPMs. Because these are all in the `proto_ipm` format, we do not need to think about technical differences between each type - we can use the exact same toolbox for analyzing both! Let's build the IPM objects and calculate deterministic per-capita growth rates!

```
ipm_list <- pdb_make_ipm(proto_list)
lambdas <- lambda(ipm_list)
```

We could now proceed with any further analyses just as we did in the case study 1. Since those types of analyses are already covered by the previous case study, we will move on to combining PADRINO data with information from other databases.

Extending analyses with other databases

Here, we show how to combine data from PADRINO with data from other external sources. Specifically, we show how to combine with data from [COMPADRE MPM database](#) and [BIEN](#), a database containing the spatial distribution and phylogenies of many plant species. We then demonstrate how we can use these combined datasets to address the question: “How do population growth rates vary by the distance of the studied population from the known range edge of that species?”

Given the question we posed above, we need to get range maps for each species and the per-capita growth rate for some populations. For the former, we will use [range maps](#) from BIEN. For the latter, we will augment PADRINO with data from COMPADRE. This analysis will not be the most complete - it is intended to demonstrate the steps for combining data, not to make a scientific point. With that in mind, let's dive in!

Required packages

BIEN allows users to download range maps programmatically from their database using the [BIEN](#) R package. You can install that from CRAN using the chunk below. We will also use `Rcompadre`, `mgcv`, `ggplot2`, `sf`, and `dplyr` to work with the data, so you'll need to install those as well.

```
install.packages(c("BIEN", "ggplot2", "sf", "dplyr", "Rcompadre", "mgcv"))
```

After that, we have to load them:

```
library(BIEN)
library(ggplot2)
library(sf)
library(dplyr)
library(Rcompadre)
library(Rpadrino)
library(mgcv)
```

Data identification

BIEN allows us to programmatically query the database and retrieve all species names for which there is a range map. we will load that, then load COMPADRE and PADRINO, and see how much overlap there is.

```
bien_rng_spps <- BIEN_ranges_list()
pdb           <- pdb_download(save = FALSE)
cdb           <- cdb_fetch("compadre")
```

```
## This is COMPADRE version 6.21.8.0 (release date Aug_20_2021)
## See user agreement at https://compadre-db.org/Help/UserAgreement
## See how to cite at https://compadre-db.org/Help/HowToCite
```

```
# Insert an underscore to make sure name format matches between COMPADRE,
# PADRINO, and BIEN
```

```
cdb_spp <- gsub(" ", "_", cdb$SpeciesAccepted)
```

```
pdb_spp <- pdb$Metadata$species_accepted
```

Nice! We have 480 overlapping species between COMPADRE/PADRINO and BIEN's range maps. This next chunk determines which species from PADRINO and COMPADRE have range maps available in BIEN:

```
all_spp <- unique(c(cdb_spp, pdb_spp))
pos_spp <- all_spp[all_spp %in% bien_rng_spps$species]

pdb_rng_spp <- unique(pdb_spp[pdb_spp %in% pos_spp])
cdb_rng_spp <- unique(cdb_spp[cdb_spp %in% pos_spp])
```

Subsetting

We probably should not use all of these, as those calculations would take quite some time for a tutorial, so we will select a subset. we will take the species for which the demographic data are from North America. For PADRINO, we need to find their `ipm_ids`, and then pass those into `pdb_subset()`. For COMPADRE, we can just use `dplyr` verbs as if we were working with a `data.frame`.

```
# First, we will create a vector of ipm_id's which meet the following requirements:
# 1. They have range maps in BIEN (species_accepted %in% pdb_rng_spp)
# 2. The model is from data collected in North America (continent == "n_america")
# 3. The data are from unmanipulated populations (treatment == "Unmanipulated")
```

```
pdb_ids <- pdb$Metadata$ipm_id[pdb$Metadata$species_accepted %in% pdb_rng_spp &
                               pdb$Metadata$continent == "n_america" &
```

```

pdb$Metadata$treatment == "Unmanipulated"]

use_pdb      <- pdb_subset(pdb, pdb_ids)

# For COMPADRE, we have to first replace "_" in the species names with a space.
# Then we can use filter() syntax to subset COMPADRE to the species we want.

cdb_rng_spp_f <- gsub("_", " ", cdb_rng_spp)

use_cdb <- filter(cdb,
  SpeciesAccepted %in% cdb_rng_spp_f &
  Continent == "N America" &
  MatrixTreatment == "Unmanipulated")

```

Check data quality

PADRINO data is validated before it is uploaded to ensure the IPM behaves as the published version behaves. There are additional checks you might want to perform on your own, and those depend on the subsequent analysis. Case study 1 shows an example of a singular kernel creating some biologically impossible results. However, there are not built-in functions in `Rpadrino` yet to assist with this. Therefore, it is usually a good idea to check the original publications just to be sure there are not caveats to the model that the authors have raised. We can find the citations using `pdb_citation()` and `pdb_report()`. `pdb_citation()` returns a character vector of citations in APA style, whereas `pdb_report()` generates an RMarkdown report based on the information in the database.

```

cites <- pdb_citations(use_pdb)

pdb_report(use_pdb)

```

we will also want to check COMPADRE for some common data issues using the `cdb_flag()` function. This is documented much more thoroughly in the [Rcompadre package website](#). For simplicity, we will just use ones which do not raise any flags, as fixing issues with COMPADRE data is beyond the scope of this case study. Furthermore, we will subset out the mean matrices, as we want to work with individual transitions.

```

cdb_f <- cdb_flag(use_cdb)

use_cdb <- filter(cdb_f, !check_NA_A & !check_NA_U & !check_NA_F & !check_NA_C &
  !check_zero_U & !check_singular_U & check_component_sum &
  check_ergodic & check_irreducible & check_primitive &
  check_surv_gte_1 & MatrixComposite == "Individual")

```

Data transformation

Next, we need to do a bit of data wrangling. From PADRINO, we only need the `ipm_id` and species names for plotting and analyzing, so we will just grab those from the metadata table. We're going to create an `sf` object for this data using the coordinates stored in the `"lat"` and `"lon"` columns of the metadata. `sf` provides a standardized interface for dealing with multiple types of spatial data, and also plays nicely with `dplyr`, which makes managing data much easier. The `st_as_sf()` function handles the conversion for us.

```

# Create a standard data.frame with ipm_id, species, lat+lon data from PADRINO

temp_coords <- use_pdb$Metadata %>%
  select(ipm_id, species_accepted, lat, lon)

```



```

# This next bit does the following:
# 1. Creates a data.frame from COMPADRE data with the same columns as PADRINO.
# 2. Changes the names so they match the PADRINO version.
# 3. Combines the COMPADRE and PADRINO versions.
# 4. Eliminates studies that do not have complete latitude/longitude information.

temp_db <- use_cdb@data %>%
  select(MatrixID, SpeciesAccepted, Lat, Lon) %>%
  setNames(names(temp_coords)) %>%
  rbind(temp_coords) %>%
  .[complete.cases(.), ]

# Finally, create an 'sf' object with the combined coordinates from COMPADRE and
# PADRINO

study_coords <- st_as_sf(temp_db,
  coords = c("lon", "lat"),
  crs = "WGS84")

```

Querying BIEN

Now that we have our final species list, we're going to download the range maps for each species using the `BIEN_ranges_load_species()` function, and then convert that into an `sf` object which will make subsequent analysis and plotting easier.

After converting the range maps to an `sf` object, we also need to create a different version of the polygons that are a set of lines representing the edges. This will allow us to quickly calculate the distance between our study points and the edge of the range. `st_cast()` handles this conversion for us.

```

# First, convert species names back to spaced version without "_"s. while the
# data.frame of species names we downloaded from BIEN before uses an "_" in
# species names, the `BIEN_ranges_load_species()` function expects names without
# underscores!

study_coords$species_accepted <- gsub("_", " ", study_coords$species_accepted)

# The next piece:
# 1. Downloads range maps
# 2. Converts each range map into an 'sf' object
# 3. Resolves issues that arise in the conversion process (e.g. self intersections)
# 4. Sorts the resulting 'sf' object alphabetically on species

rng_maps <- BIEN_ranges_load_species(study_coords$species_accepted) %>%
  st_as_sf() %>%
  st_make_valid() %>%
  arrange(species)

# We need to create a copy of each range map that, rather than a polygon format,
# is a line format. This enables us to use functions to compute distance to edge
# much more easily.

line_maps <- st_cast(rng_maps, "MULTILINESTRING")

```

```

# Put the "_"'s back into study_coords so that we can match all names later on.

study_coords$species_accepted <- gsub(" ", "_", study_coords$species_accepted)

# Finally, sort study_coords alphabetically as well. Now, the species_accepted
# column should be identical to the species column in rng_maps. This will be
# important in the next step.

study_coords <- arrange(study_coords, species_accepted) %>%
  .[!duplicated(.$species_accepted), ]

```

Compute distance from edges

Ok, we're finally ready to compute the distance from each study site to the range edge. We're going to use the `st_distance()` function for this. This finds the minimum distance between the first and second arguments and computes a matrix for all possible combinations. It will ignore the fact that sometimes the closest edge is an ocean (which our species cannot grow in). However, working out how to improve that calculation is a problem for another day!

We start by extracting a distance matrix and taking the diagonal. The diagonal represents the shortest distance between our species study site and the edge of the polygon of its range map (NB: This only works because we sorted each object alphabetically ahead of time!). Next, we add in the species name information and set the data frame's names to something useful. Finally, we will convert the distances to kilometers.

```

# Quickly check to make sure all of our species line up positionally.
# If not, we'd need to make sure they do, otherwise it will be difficult
# to extract the distances from the distance matrix we are about to compute!

stopifnot(all(line_maps$species == study_coords$species_accepted))

# This next piece does:
# 1. Computes distances between all pairs of points and line objects
# 2. Extracts the diagonal of the distance matrix. This represents the distance
# from a species' study site to the edge of its range (again, only because
# we sorted study_coords and line_maps alphabetically).
# 3. converts this information to a data.frame
# 4. Adds the species names to that data.frame
# 5. Sets column names for that data.frame
# 6. Converts the distance to km

dist_from_edge <- st_distance(study_coords, line_maps) %>%
  diag() %>%
  data.frame() %>%
  cbind(study_coords$species_accepted, .) %>%
  setNames(c("species", "distance_in_meters")) %>%
  mutate(
    distance_in_km = round(as.numeric(distance_in_meters) / 1e3, 2)
  )

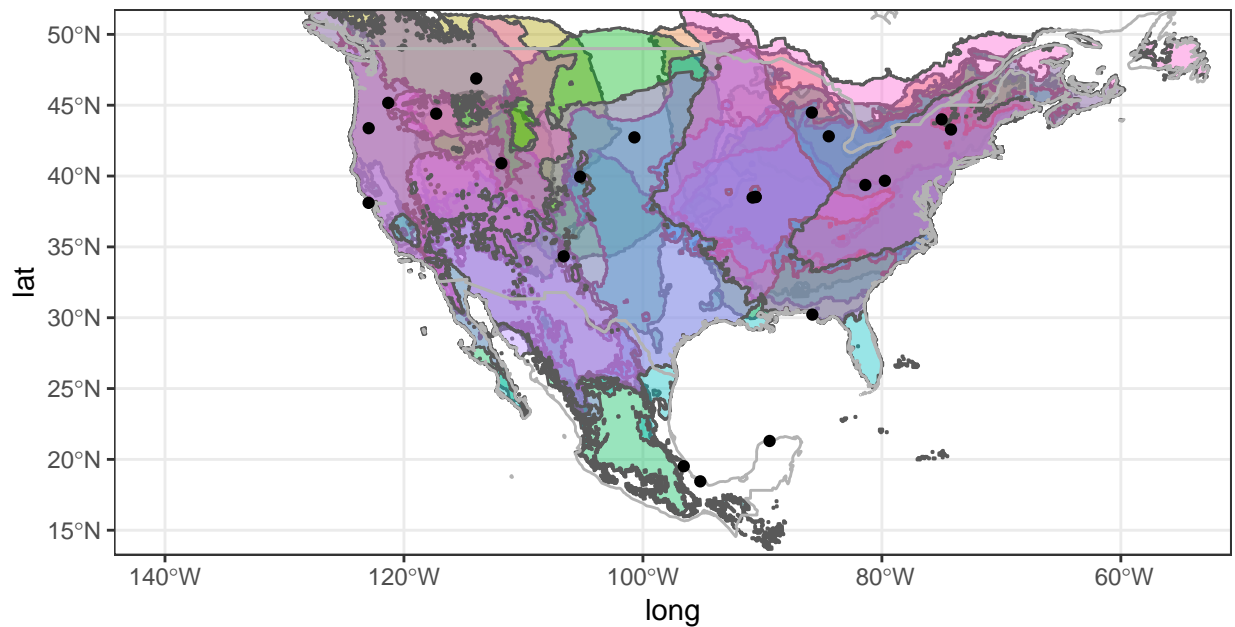
```

Visualize our dataset

we will plot our range maps with the study sites overlaid on them using `ggplot2`. `ggplot2` has built in geoms designed to handle `sf` objects, which will make our lives much easier!

```
world      <- map_data("world")
n_america  <- filter(world, region %in% c("USA", "Canada", "Mexico"))

ggplot(rng_maps) +
  geom_sf(aes(fill = species), alpha = 0.4) +
  geom_polygon(data = n_america, aes(x = long, y = lat, group = group),
              inherit.aes = FALSE,
              color = "grey70",
              fill = NA) +
  geom_sf(data = study_coords) +
  coord_sf(xlim = c(-140, -55),
           ylim = c(15, 50)) +
  theme_bw() +
  theme(
    legend.position = "none"
  )
```



Already, we can see that our range maps do not perfectly align with the COMPADRE and PADRINO population coordinates. We can check and see which study populations are actually contained by their range map like so:

```
# Notice that now we are using the POLYGONS object (rng_maps) as opposed to the
# to the LINESTRING version (line_maps).
```

```
covered_ind <- st_covered_by(study_coords,
                             st_make_valid(rng_maps),
                             sparse = FALSE) %>%
  diag()
```

```
# Print studies not covered by BIEN range map
study_coords[!covered_ind , ]
```

```
## Simple feature collection with 6 features and 2 fields
## Geometry type: POINT
## Dimension:      XY
## Bounding box:   xmin: -122.9567 ymin: 18.45 xmax: -85.82 ymax: 45.16667
## Geodetic CRS:   WGS 84
## # A tibble: 6 x 3
##   ipm_id species_accepted      geometry
##   <chr>   <chr>              <POINT [°]>
## 1 241875 Astragalus_tyghensis (-121.3167 45.16667)
## 2 242052 Calathea_ovandensis  (-95.2 18.45)
## 3 239708 Euphorbia_telephioides (-85.82 30.21972)
## 4 244332 Lupinus_tidestromii   (-122.9567 38.10861)
## 5 247007 Mammillaria_gaumeri   (-89.4 21.3)
## 6 241182 Tillandsia_deppeana   (-96.58333 19.51667)
```

These are COMPADRE matrices. Rather than try to figure out what's going on, we will just drop those out of our analysis.

```
study_coords <- study_coords[covered_ind, ]
```

Compute lambdas for each type of model

Great! The next step is to generate and then join our lambda values with the distance information. This is a two-step process. First, we will build our PADRINO IPMs:

```
# Extract PADRINO IDs - we do not want to give COMPADRE ones to PADRINO machinery!
```

```
pdb_ids <- study_coords$ipm_id[study_coords$ipm_id %in% pdb$Metadata$ipm_id]
```

```
# Construct the proto_ipm list
```

```
proto_list <- pdb_make_proto_ipm(
  use_pdb,
  pdb_ids
)
```

```
## 'ipm_id' aaa310 has the following notes that require your attention:
```

```
## aaa310: 'Geo and time info retrieved from COMPADRE (v.X.X.X.4)'
```

```
## 'ipm_id' aaa329 has the following notes that require your attention:
```

```
## aaa329: 'Based on IPM from Rose Ecology 2005; The GPS coordinates were approximated
## to the closest geographic location described in the reference'
```

```
## 'ipm_id' aaa385 has the following notes that require your attention:
```

```
## aaa385: 'Same data as AAA385. State variable Height (Cm)'
```

```

# Construct the IPMs

ipm_list <- pdb_make_ipm(proto_list)

# Some IPMs may have many values for lambda, because they were constructed from
# vital rate models that have time varying parameters (e.g. random effects for
# year). we will need to account for this. We need to convert those from a list to
# a data.frame for modeling, and need to keep track of which lambda belongs to
# which ID. The loop below will correctly format this.

lambdas <- lambda(ipm_list, type_lambda = "last")

temp <- data.frame(ipm_id = NA,
                  lambda = NA)

for(i in seq_along(lambdas)) {

  # Create a temporary object to store lambda values and ipm_id's. Each lambda
  # value will have its own row, with the corresponding ipm_id next to it. This
  # will help us track which value belongs to which model.

  temp_2 <- data.frame(ipm_id = names(lambdas)[i],
                      lambda = lambdas[[i]])

  # I don't normally recommend using rbind in a 'for' loop, but there aren't many
  # iterations here, so we will not worry about the memory footprint
  temp <- rbind(temp, temp_2)

}

# Remove the dummy row of NAs

temp <- temp[-1, ]

```

Next, we will get our COMPADRE lambdas, and stick them back in with the PADRINO lambdas.

```

use_cdb <- filter(use_cdb, MatrixID %in% study_coords$ipm_id)

matAs <- matA(use_cdb)

use_cdb@data$lambda <- vapply(matAs,
                             function(x) Re(eigen(x)$values[1]),
                             numeric(1L))

cdb_lambda <- use_cdb@data %>%
  select(MatrixID, lambda) %>%
  setNames(c("ipm_id", "lambda"))

all_lambdas <- rbind(temp, cdb_lambda)

```

Finally, we need to join lambda values with coordinate data set to recover the species names, and then use those to join with the distance from edge object. Once that's done, we can plot everything!

```
all_lambdas <- left_join(all_lambdas, study_coords, by = "ipm_id") %>%
  select(-geometry)

all_data <- left_join(all_lambdas, dist_from_edge,
  by = c("species_accepted" = "species"))
```

Regression modelling

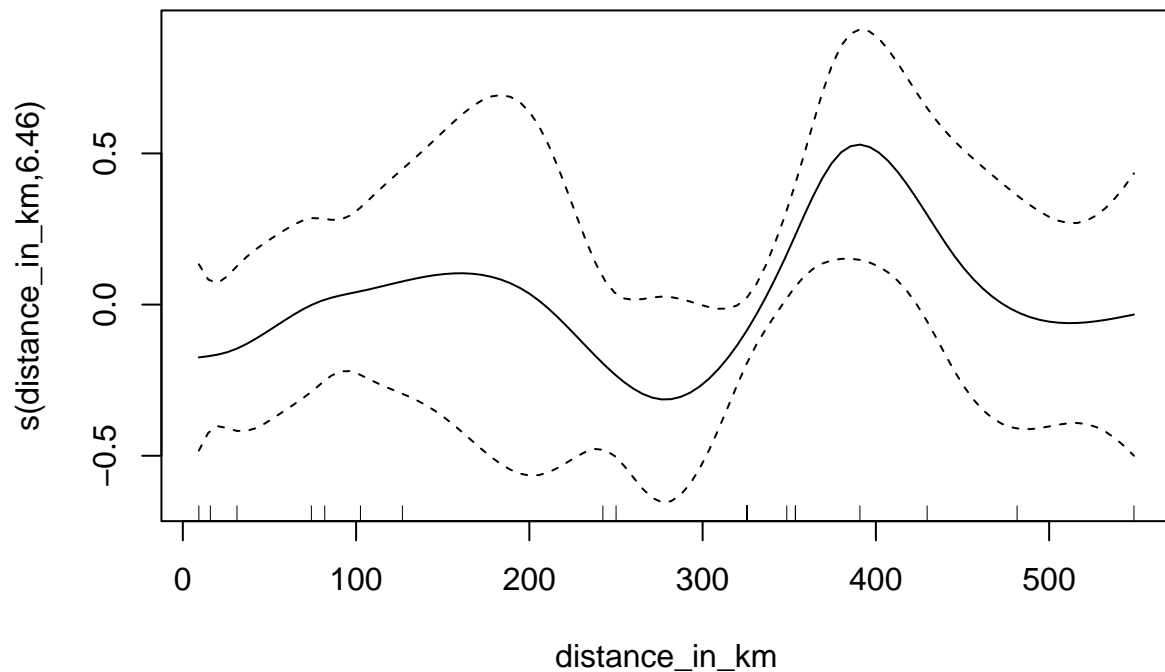
We're ready to plot and analyze the data. GAMs (Wood 2011) are a great way to spot general trends in data, so we will use those.

```
lambda_by_dist <- gam(lambda ~ s(distance_in_km, bs = "cs"),
  data = all_data,
  family = Gamma(link = "identity"))

summary(lambda_by_dist)

##
## Family: Gamma
## Link function: identity
##
## Formula:
## lambda ~ s(distance_in_km, bs = "cs")
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1.13381    0.04843   23.41 9.22e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##             edf Ref.df      F p-value
## s(distance_in_km) 6.463     9 1.305  0.135
##
## R-sq.(adj) =  0.197   Deviance explained = 47.5%
## GCV = 0.059745   Scale est. = 0.048724   n = 27

plot(lambda_by_dist)
```



```

preds <- cbind(data.frame(predict(lambda_by_dist,
                                data.frame(distance_in_km = seq(0, 550, 1)),
                                type = "response",
                                se.fit = TRUE)),
               x = seq(0, 550, 1)) %>%
mutate(upper = fit + se.fit * 1.96,
       lower = fit - se.fit * 1.96)

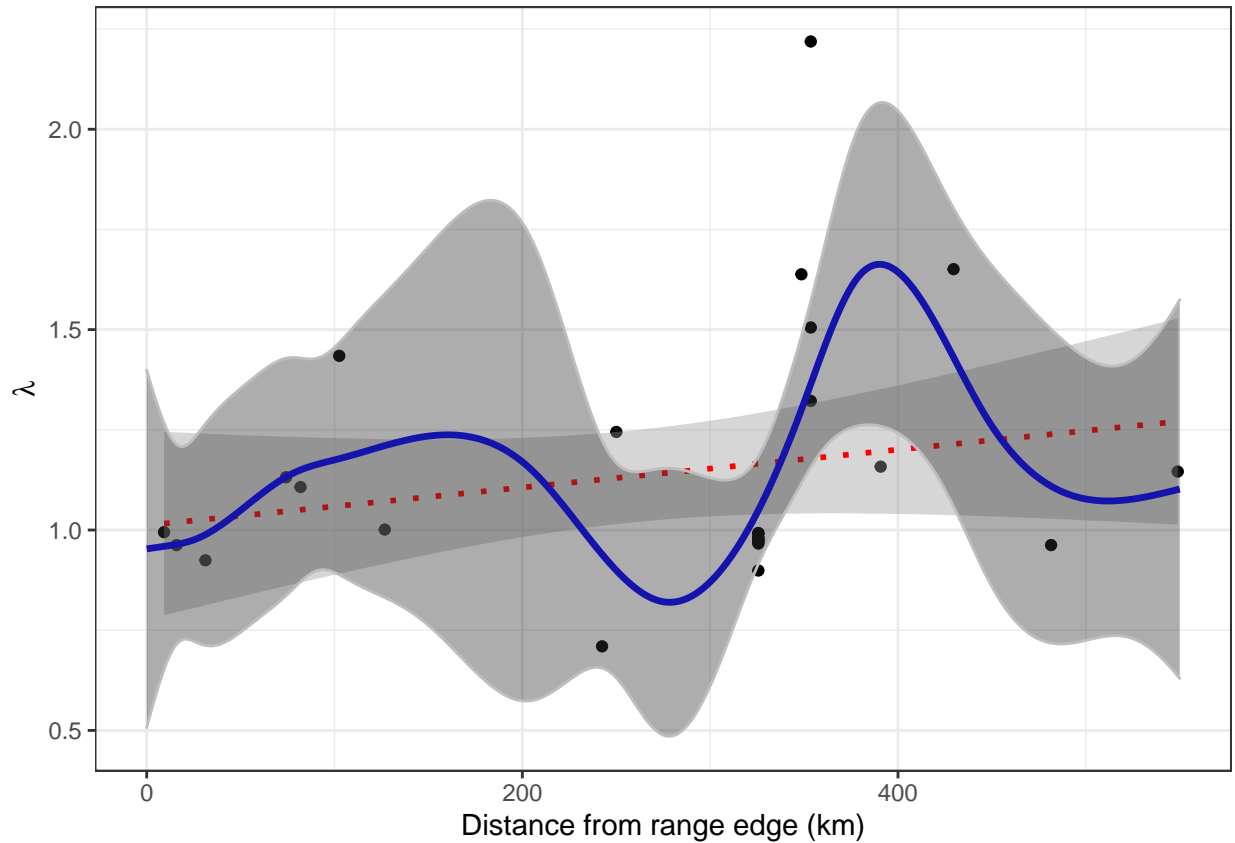
ggplot(all_data, aes(x = distance_in_km, y = lambda)) +
  geom_point() +
  geom_smooth(method = "glm",
             formula = y ~ x,
             method.args = list(family = Gamma("identity")),
             color = "red",
             linetype = "dotted") +
  geom_line(data = preds,
           aes(x = x, y = fit),
           inherit.aes = FALSE,
           color = "blue",
           size = 1.2) +
  geom_ribbon(data = preds,
            aes(x = x, ymin = lower, ymax = upper),
            color = "grey",
            alpha = 0.4,

```

```

    inherit.aes = FALSE) +
  theme_bw() +
  xlab("Distance from range edge (km)") +
  ylab(expression(lambda))

```



There is a positive trend in range centrality and species performance (red line), and the GAM is likely overfit (blue line). There is a lot of residual variance, and we can certainly find better ways to model this phenomenon, but this is a good start for an exploratory analysis. We will leave the further analyses as an exercise to you!

Recap

We have shown how to combine PADRINO data with user-defined IPMs as well as join it with information from other databases. This is hardly a comprehensive overview of PADRINO's applications - there are many other uses and databases one could combine PADRINO with. It is our hope that this and the previous case study provide a general guide to the considerations and steps one needs to take when using this data!

Appendix 6: Supplementary Information for Chapter 2

Database schema

PADRINO is structured such that each model gets one row for the Metadata table, and an arbitrary number of rows for every table after that. Some models may have 0, 1, or many rows for some of these tables. Information for each model is linked across tables by the `ipm_id` column. Complete descriptions of each column are provided [here](#).

Metadata <ul style="list-style-type: none">• <code>ipm_id</code>• <code>species_author</code>• <code>species_accepted</code>• <code>tax_genus</code>• <code>tax_order</code>• <code>tax_class</code>• <code>tax_phylum</code>• <code>kingdom</code>• <code>organism_type</code>• <code>dicot_monocot</code>• <code>angio_gymno</code>• <code>authors</code>• <code>journal</code>• <code>pub_year</code>• <code>doi</code>• <code>corresponding_author</code>• <code>email_year</code>• <code>remark</code>• <code>apa_citation</code>• <code>demog_appendix_link</code>• <code>duration</code>• <code>start_year</code>• <code>start_month</code>• <code>end_year</code>• <code>end_month</code>• <code>periodicity</code>• <code>population_name</code>• <code>number_publications</code>• <code>lat</code>• <code>lon</code>• <code>altitude</code>• <code>country</code>• <code>continent</code>• <code>ecoregion</code>• <code>studied_sex</code>• <code>evict_used</code>• <code>evict_type</code>• <code>treatment</code>• <code>has_time_lag</code>• <code>has_age</code>• <code>has_dd</code>• <code>is_periodic</code>	StateVariables <ul style="list-style-type: none">• <code>ipm_id</code>• <code>state_variable</code>• <code>discrete</code> ContinuousDomains <ul style="list-style-type: none">• <code>ipm_id</code>• <code>state_variable</code>• <code>domain</code>• <code>lower</code>• <code>upper</code>• <code>kernel_id</code>• <code>notes</code> IntegrationRules <ul style="list-style-type: none">• <code>ipm_id</code>• <code>state_variable</code>• <code>domain</code>• <code>n_meshpoints</code>• <code>integration_rule</code>• <code>kernel_id</code> StateVectors <ul style="list-style-type: none">• <code>ipm_id</code>• <code>expression</code>• <code>n_bins</code>• <code>comment</code> IpmKernels <ul style="list-style-type: none">• <code>ipm_id</code>• <code>kernel_id</code>• <code>formula</code>• <code>model_family</code>• <code>domain_start</code>• <code>domain_end</code> VitalRateExpr <ul style="list-style-type: none">• <code>ipm_id</code>• <code>demographic_parameter</code>• <code>formula</code>• <code>model_type</code>• <code>kernel_id</code>	ParameterValues <ul style="list-style-type: none">• <code>ipm_id</code>• <code>demographic_parameter</code>• <code>state_variable</code>• <code>parameter_name</code>• <code>parameter_value</code> EnvironmentalVariables <ul style="list-style-type: none">• <code>ipm_id</code>• <code>env_variable</code>• <code>vr_expr_name</code>• <code>env_range</code>• <code>env_function</code>• <code>model_type</code> ParSetIndices <ul style="list-style-type: none">• <code>ipm_id</code>• <code>env_variable</code>• <code>vr_expr_name</code>• <code>range</code>• <code>kernel_id</code>• <code>drop_levels</code>
--	--	---

Eigenständigkeitserklärung

Hiermit erkläre ich, dass die Arbeit mit dem Titel “Disentangling the mechanisms underlying the island species-area relationship (ISAR)” bisher weder bei der Naturwissenschaftlichen Fakultät III Agrar und Ernährungswissenschaften, Geowissenschaften und Informatik der Martin-Luther-Universität Halle-Wittenberg noch einer anderen wissenschaftlichen Einrichtung zum Zweck der Promotion vorgelegt wurde.

Ferner erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst sowie keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die den Werken wörtlich oder inhaltlich entnommenen Stellen wurden als solche von mir kenntlich gemacht. Ich erkläre weiterhin, dass ich mich bisher noch nie um einen Doktorgrad beworben habe.

Halle (Saale), den 18.01.2022

Sam C. Levin