Stand-alone analyses with PADRINO and RPadrino

Contents

PADRINO and RPADRINO

PADRINO's syntax is rather opaque, requiring a variety of transformations to make it at all useful to most users. To that end, we've created *RPadrino* to streamline the process of interacting with it from *R*. At some point, it would be nice to create bindings to Julia and Python, which would further enhance accessibility. Alas, I have to finish a PhD soon and so those will have to wait a little longer. Now, on to the actual tutorial!

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 housed here 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 don't already have it, install with:

```r
install.packages("dplyr")
```

Next, we'll show how to compute sensitivity and elasticity for simple models, as well as derive a few life history traits 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'll define a couple of our own to help tie it all together.

For this example, we'll 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). 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.

Subsetting using Metadata and other tables

We can find those using a combination of *dplyr* (Wickham et al. 2021) and *RPadrino* code. *RPadrino* provides the `pdb_subset` function, but this currently only takes `ipm_id`s that we want to keep. The functionality will get expanded, but it's surprisingly complicated to manage that in a user-friendly interface, so we're stuck with this for now. This means that we have to work out which `ipm_id`s correspond to the models we want, and then pass those to `pdb_subset()`.

```r
library(dplyr)
library(RPadrino)

pdb <- pdb_download(save = FALSE)

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, quite a few of these may be stochastic models, as well as density-dependent models. We'll want to get rid of those too.

```r
stoch_ind <- unique(c(simple_pdb$EnvironmentalVariables$ipm_id,
                      simple_pdb$ParSetIndices$ipm_id,
                      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. We can do a quick sanity check to make sure our subsetting produced only these kernels like so:

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

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

Great, all `P`s and `F`s! Finally, we'll remove the duplicated species names so that we have a small-ish data set. 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 subsetted, 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 next case study.

RPadrino provides the `pdb_make_proto_ipm()` function. This takes a `pdb` object and produces a list of `proto_ipm`s. There are additional options that we can pass to this, but we'll ignore those for now, and just focus on creating and understanding what the outputs are.

```
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 Â°C, and constant survival probability
## of large pike'

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

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 isn't 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 model is cobbled together from disparate sources, Shelley 1818). Thus, the authors have cobbled together an IPM with disparate data sources. 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).

We'll 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
## Podarcis lilfordi
## Nerodia sipedon
##
## You can inspect each model by printing it individually.
```

```
simple_det_list$aaaa34
```

```
## 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"
##  [9] "fn_b"   "fn_m"   "t_r"    "d_mean" "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)
```

```r
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"    
##  [9] "bR"     "svlM"   "muD"    "sigmaD"
## 
## 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 to produce new trait distributions. We can now build the actual IPM objects. We'll also check for convergence to asymptotic dynamics using the `is_conv_to_asymptotic` function.

```r
all_ipms <- pdb_make_ipm(simple_det_list)

check_conv <- is_conv_to_asymptotic(all_ipms)
```

```
## aaa310, aaa341, ccccc1, ddddd3, ddddd5, dddd10, dddd30 did not converge!
```

We can see that a few of these need more than the default number of iterations to converge to asymptotic dynamics. Since we need correct lambda values to compute elasticity, we'll need to re-run those models.

`pdb_make_ipm()` contains the `addl_args` argument that tells the function how to deviate from the default behavior of `ipmr::make_ipm()`. It's accepts nested lists with the following format:

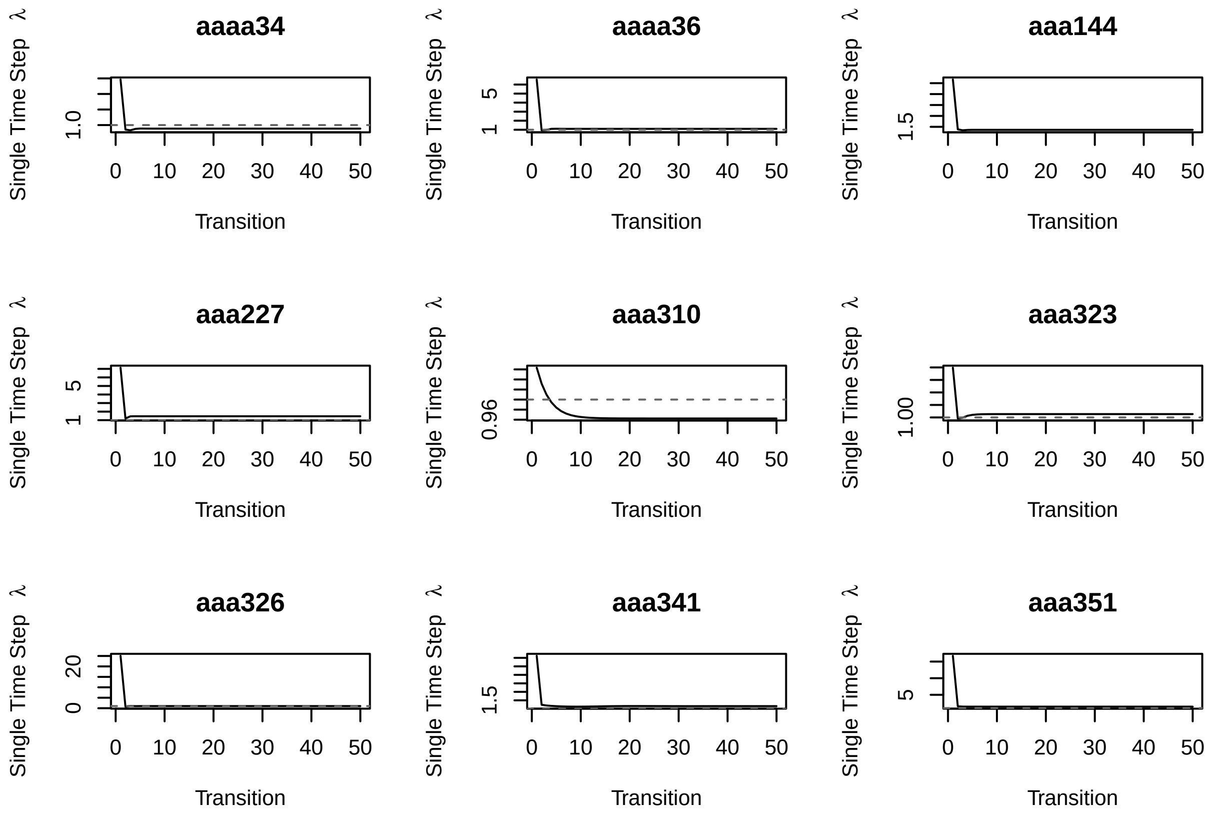
```
list(<ipm_id_1> = list(<make_ipm_arg_name_1> = <XXX>,
                       <make_ipm_arg_name_2> = <YYY>),
     <ipm_id_2> = list(<make_ipm_arg_name_1> = <XXX>,
                       <make_ipm_arg_name_5> = <ZZZ>))
```

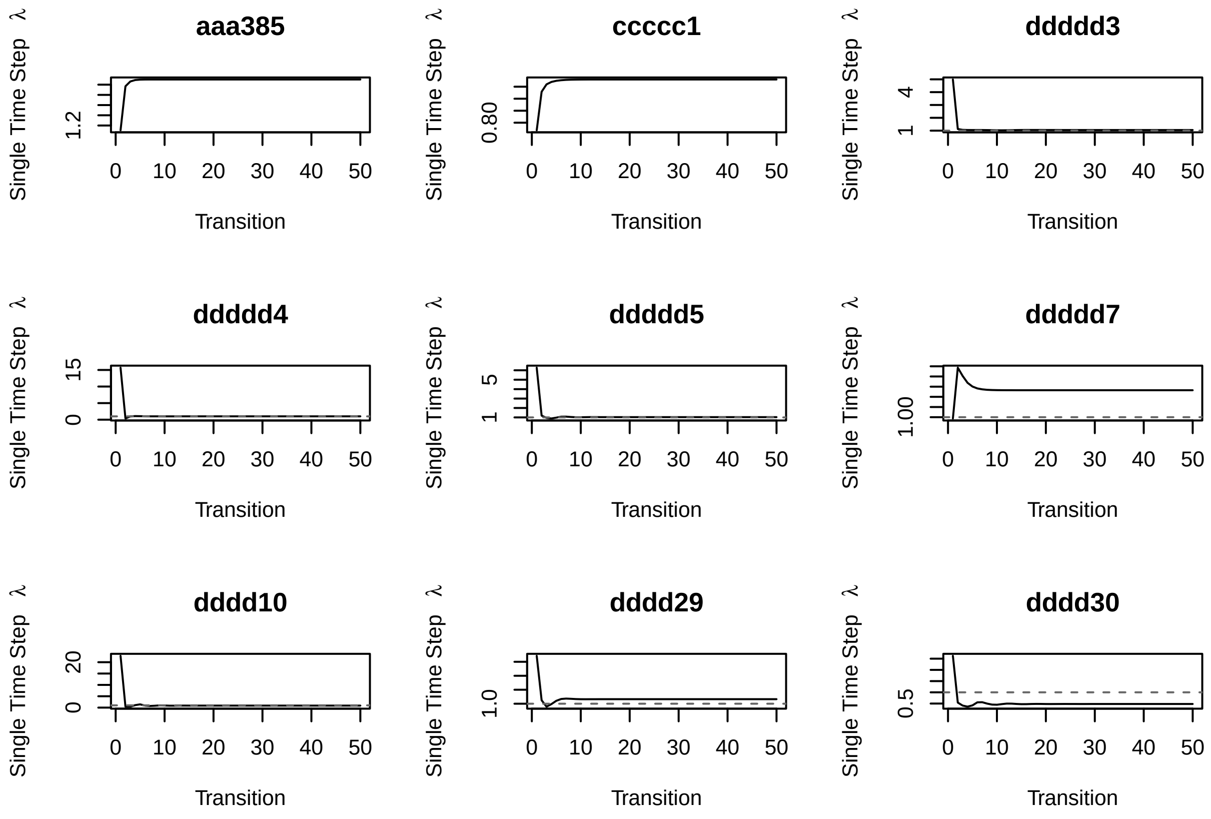
We replace the values in `<>` with the actual `ipm_id`s, argument names, and values we want them to have. We can do this many models a bit more concisely:

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

ind_conv <- c("aaa310",
              paste0("ddddd", c(3,5)),
              paste0("dddd", c(10, 24, 26, 30)))

arg_ind <- vector("list", length = length(ind_conv)) %>%
  setNames(names(ind_conv))

# Next, we set create an entry in each list with iterations = <some number>
# We'll use 250 for this example

arg_list <- lapply(arg_ind,
                   function(x, n_iter) list(iterations = n_iter),
                   n_iter = 250)

new_ipms <- pdb_make_ipm(simple_det_list, addl_args = arg_list)

check_conv <- is_conv_to_asymptotic(new_ipms)
```

```
## aaa310, aaa341, ccccc1, ddddd3, ddddd5, dddd10, dddd30 did not converge!
```

```
# We can also plot the lambda time series using conv_plot methods for pdb_ipms
par(mfrow = c(3, 3))

conv_plot(new_ipms)
```


aaaa34
aaaa36
aaa144
aaa227
aaa310
aaa323
aaa326
aaa341
aaa351
Single Time Step λ
Transition
1.0
1
5
1.5
0.96
1.00
0
20
0 10 20 30 40 50



```
# Note that the y-axis values are *very* small. While it hasn't technically
# converged, we can probably use these lambda values safely for elasticity
# analysis.
```

Further analyses

RPadrino contains methods for some *ipmr* functions. These include `lambda`, `left_ev`, and `right_ev`. We need all three of these to compute sensitivity and elasticity. We also need the mesh binwidth so we can perform integrations. *ipmr* 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)
```

```
## 'x' did not converge to asymptotic dynamics after 51 iterations.
## Will re-iterate the model 100 times and check for convergence.

## model is now converged :)
```

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

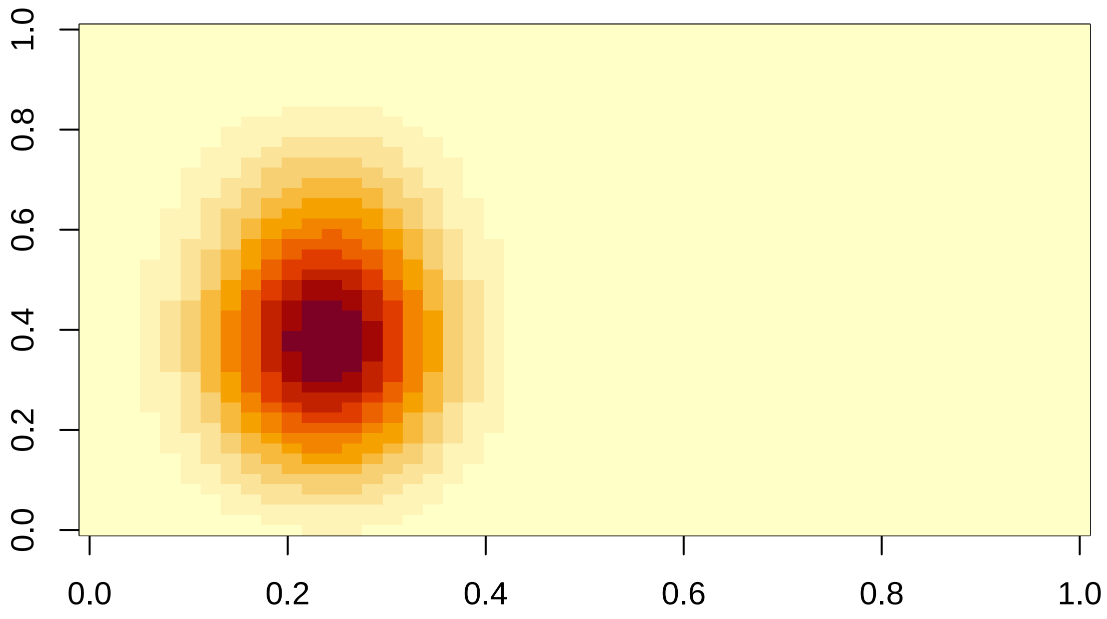
Sensitivity

With these, we can now compute sensitivity. This is given by $\mathbf{s}(z_0', z_0) = \frac{v(z_0')w(z_0)}{\langle v,w \rangle}$, 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'll use `lapply(seq_along())` to iterate over each model.

```r
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()`:

```r
image(t(sens_list$aaa385))
```



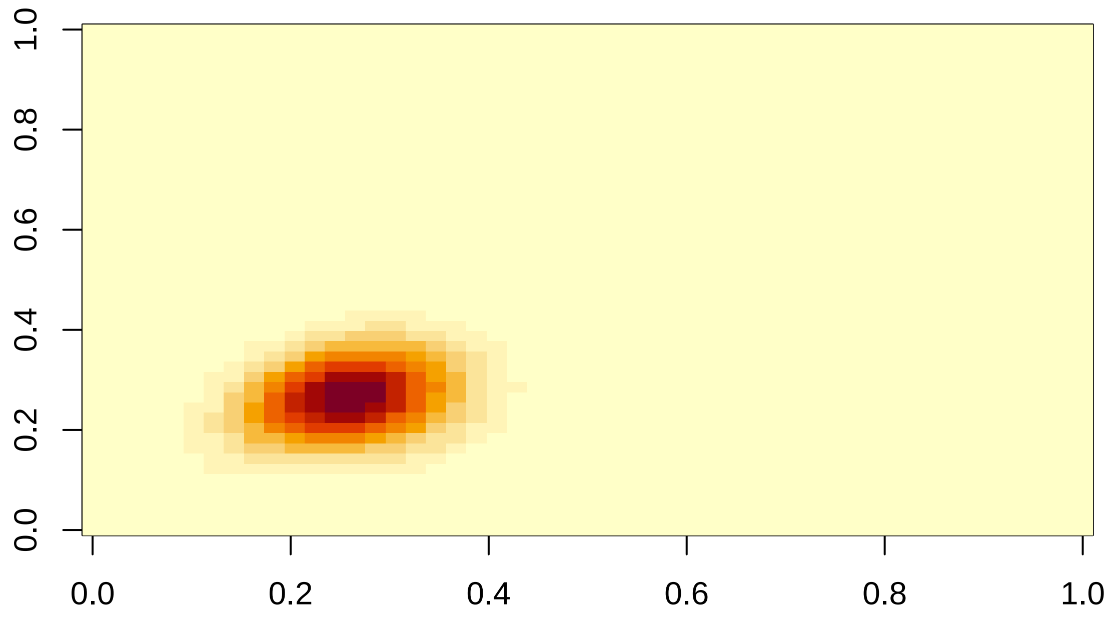
Elasticity

We can compute elasticity without much more effort. We need one more piece of information from the IPM list that we haven't extracted - the iteration kernel. We can get those by `lapply`ing *ipmr*'s `make_iter_kernel()` to 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 <- lapply(new_ipms, make_iter_kernel)

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

This is defined as $\bar{r}(z_0) = eFN$. F is the fecundity kernel, and we can get that from each IPM in our list using:

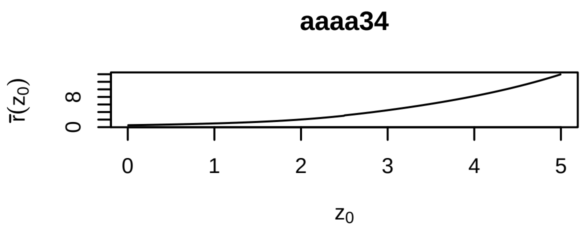
```r
F_kerns <- lapply(new_ipms, function(x) x$sub_kernels$F)
```

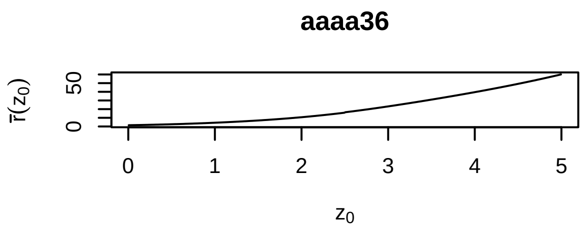
$N(z', z_0)$ is the fundamental operator. This tells us the expected amount of time an individual will spend in state z' given an initial state z_0. The fundamental operator is defined as $(I - P)^{-1}$ (see Ellner, Childs, & Rees 2016 Chapter 3 for the derivation of this). I is an identity kernel, with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

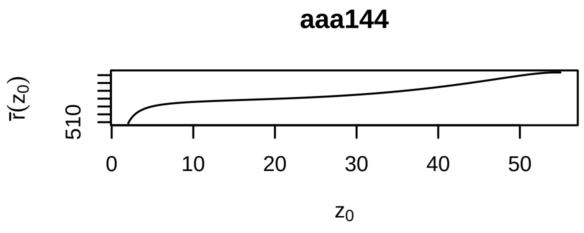
```r
# Function to create an identity matrix with dimension equal to P
make_i <- function(P) {
  return(
    diag(nrow(P))
  )
}

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

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

  solve(I - P)
})
```

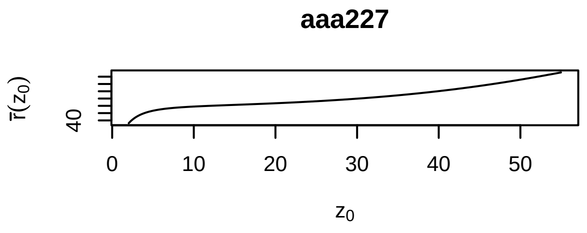
e is a constant function $e(z) \equiv 1$. In practice, the left multiplication of eF has the effect of computing the column sums of F. We'll replace the e with a call to `colSums()` in our code below (this will run faster than doing the multiplication). We now have everything we need to compute and visualize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

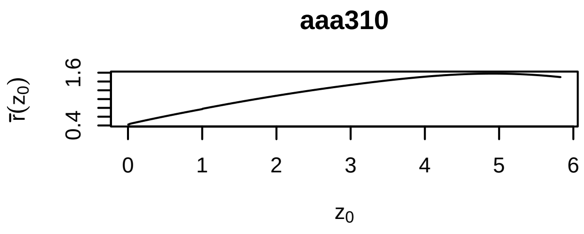
```
# We wrap the computation in as.vector so that it returns a simple numeric vector
# rather than a 1 x N matrix

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'll 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(3, 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]))

}
```

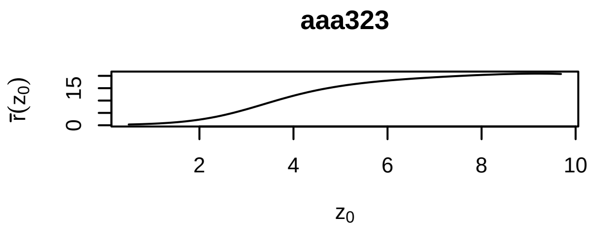

aaaa34
$\bar{r}(z_0)$
0
8
0 1 2 3 4 5
z_0

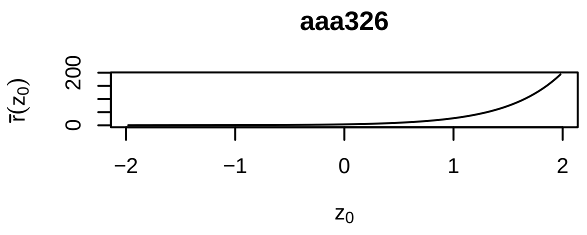

aaaa36
$\bar{r}(z_0)$
0
50
0 1 2 3 4 5
z_0

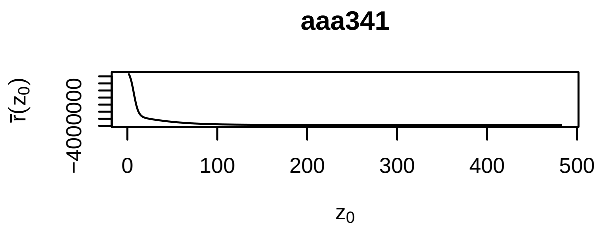

aaa144
$\bar{r}(z_0)$
510
0 10 20 30 40 50
z_0

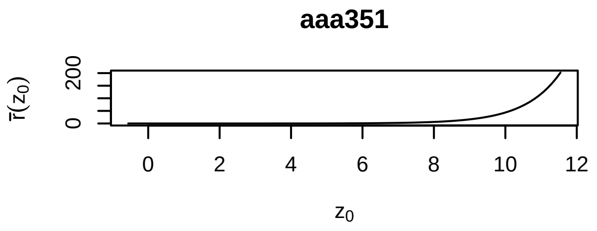

aaa227
$\bar{r}(z_0)$
40
0 10 20 30 40 50
z_0

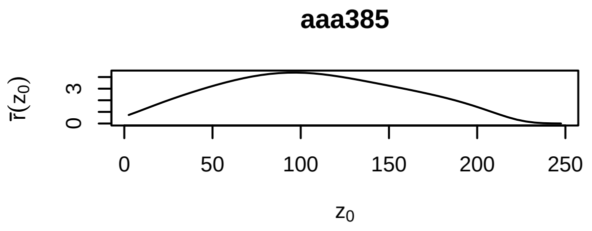

aaa310
$\bar{r}(z_0)$
0.4
1.6
0 1 2 3 4 5 6
z_0

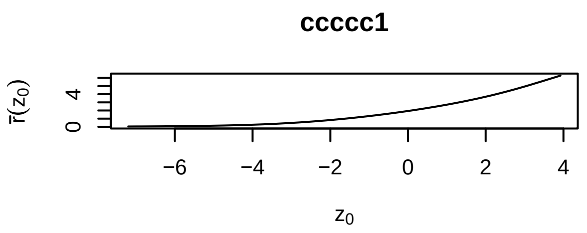

aaa323
$\bar{r}(z_0)$
0
15
2 4 6 8 10
z_0

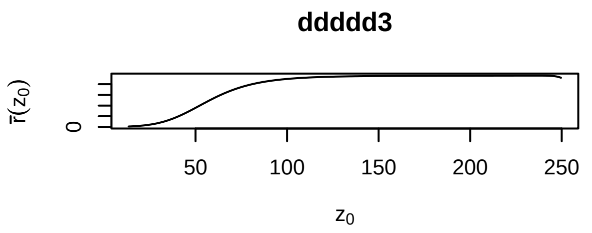

aaa326
$\bar{r}(z_0)$
0
200
−2
−1
0
1
2
z_0

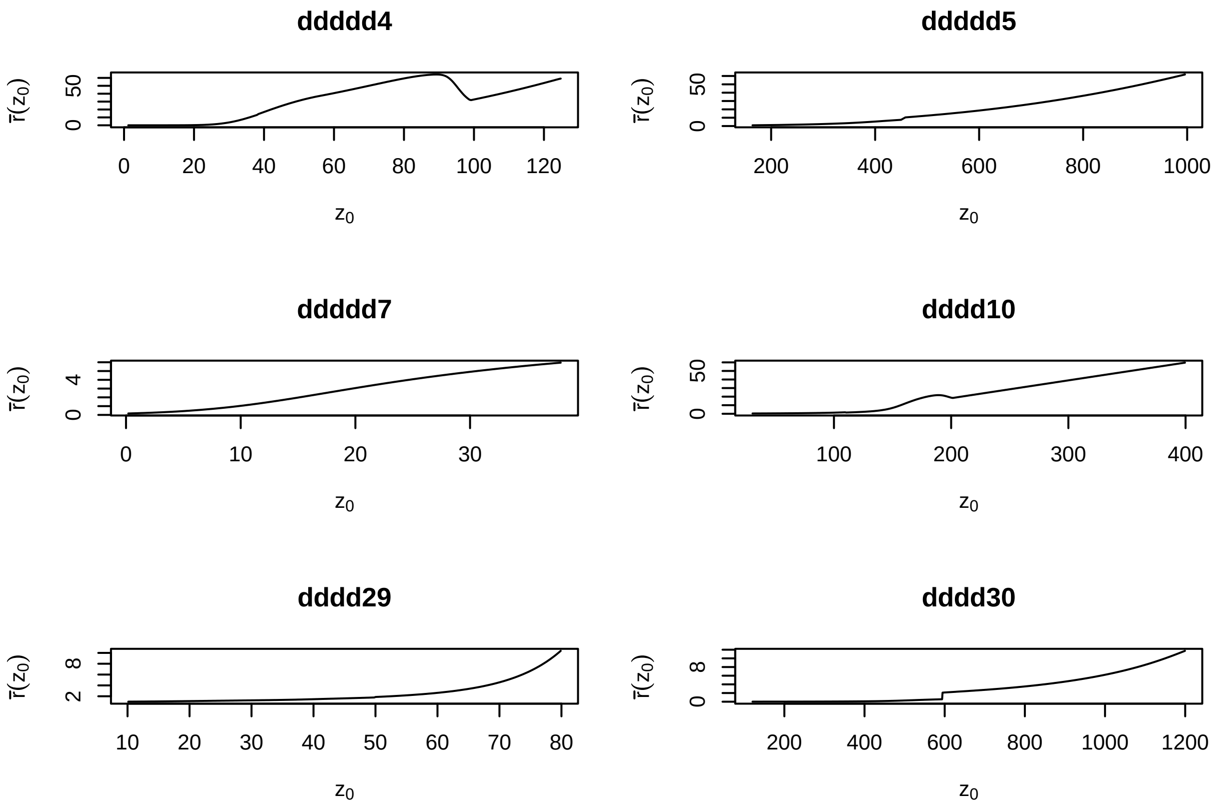

aaa341
$\bar{r}(z_0)$
−4000000
0
100
200
300
400
500
z_0


aaa351
$\bar{r}(z_0)$
0
200
0
2
4
6
8
10
12
z_0


aaa385
$\bar{r}(z_0)$
0
3
0
50
100
150
200
250
z_0


cccccc1
$\bar{r}(z_0)$
0
4
−6
−4
−2
0
2
4
z_0


ddddd3
$\bar{r}(z_0)$
0
50
100
150
200
250
z_0



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 comprehensive descriptions of them in PADRINO. With this caveat in mind, we'll 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 additinal effort, but are usually worth it. We'll step through the code pieces required to extract these below. We'll start with an example computing the sensitivity of λ to vital rate function values. After that, we'll 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 formula for the kernel, and so no general formula exists. However, we can use the chain rule to work out what it should be. The general formula for a given perturbation is:

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

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'll 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

In order to keep things simple, we'll 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_paratemer` 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 easy, we'll just use the first 3 in this index

keep_ind <- keep_ind[1:3]

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

Next, we'll 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 * 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"
##  [9] "fn_b"   "fn_m"   "t_r"    "d_mean" "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)
```

```
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 * 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"
##  [9] "fn_b"   "fn_m"   "t_r"    "d_mean" "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)
```

```
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"
##  [7] "g_i"       "g_el"      "g_frost"   "g_s"       "g_sd"      "fl_i"
## [13] "fl_s"      "germ_i"    "germ_el"   "es_p"      "sdl_mean"  "sdl_sd"
## [19] "infl_n"    "fl_n"      "seed_i"    "smlv"      "sslv"      "wmlv"
## [25] "wslv"      "fmlv"      "fslv"      "annfrost"  "cumfrost"  "elev_germ"
## [31] "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 each $P = s(z) * G(z', z)$. In the third models, $s(z)$ is comprised of two additional functions, which we could perturb individually. We'll show a quick example of that after applying our perturbation kernels to all 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 zs and z's for brevity). The $\circ$ denotes point-wise multiplication.

Implement the models

Now that we've 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

Next, we need to implement the formula above. This is fairly straightforward, and we'll make use of another function in *RPadrino*: `vital_rate_funs()`. This extracts the vital rate function values from each model and returns them in a named list. 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: 3.6806
## 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
```

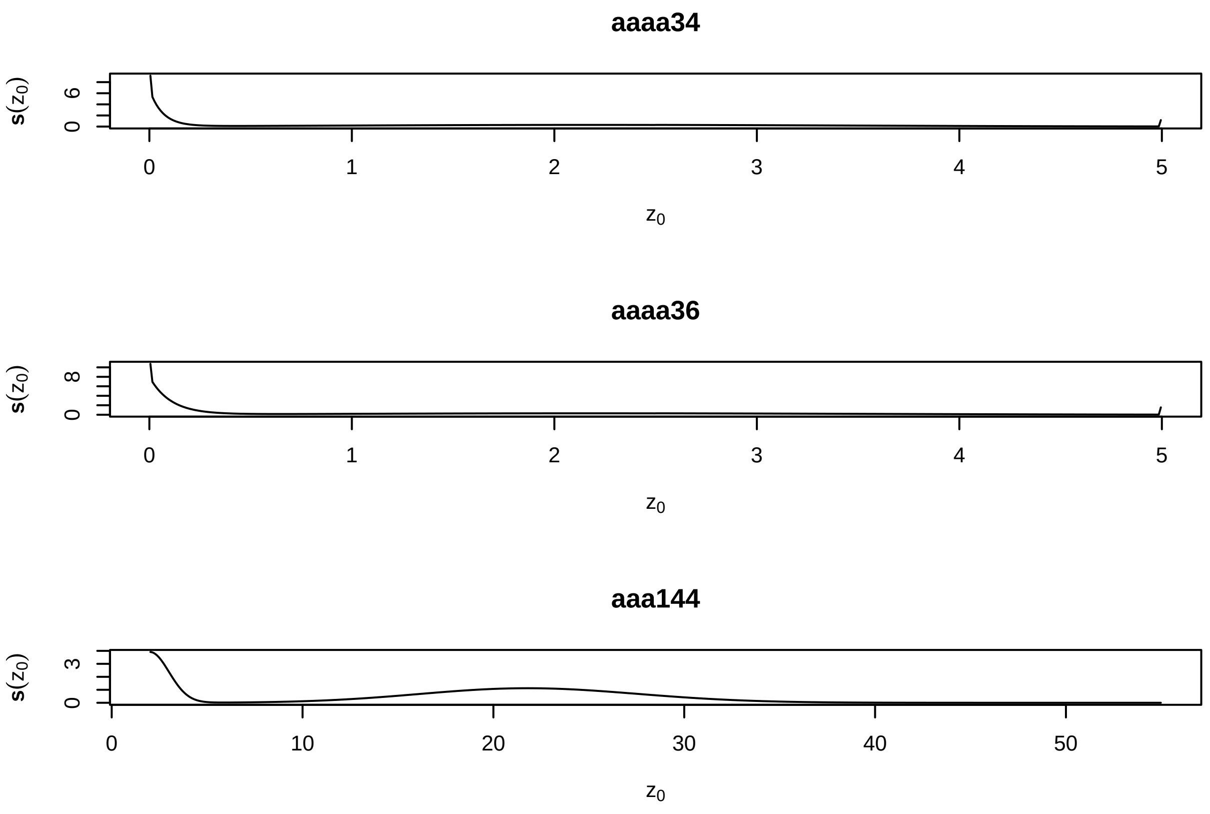
```
## $P
## s (not yet discretized): A 1000 x 1000 kernel with minimum value: 0.9712 and maximum value: 0.9997
## ssurv (not yet discretized): A 1000 x 1000 kernel with minimum value: 0.9729 and maximum value: 0.99
## wsurv (not yet discretized): A 1000 x 1000 kernel with minimum value: 0.9983 and maximum 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 won't always be true for PADRINO, so care must be taken at this step to make sure you extract the correct values!

Note that we also need the value of dz to implement the denominator. We'll 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) {
                      G <- vr_funs[[idx]]$P$g
                      v <- unlist(l_evs[[idx]])
                      w <- unlist(r_evs[[idx]])
                      d_z <- d_zs[[idx]]

                      numerator <- left_mult(G, v) * 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]))

}
```

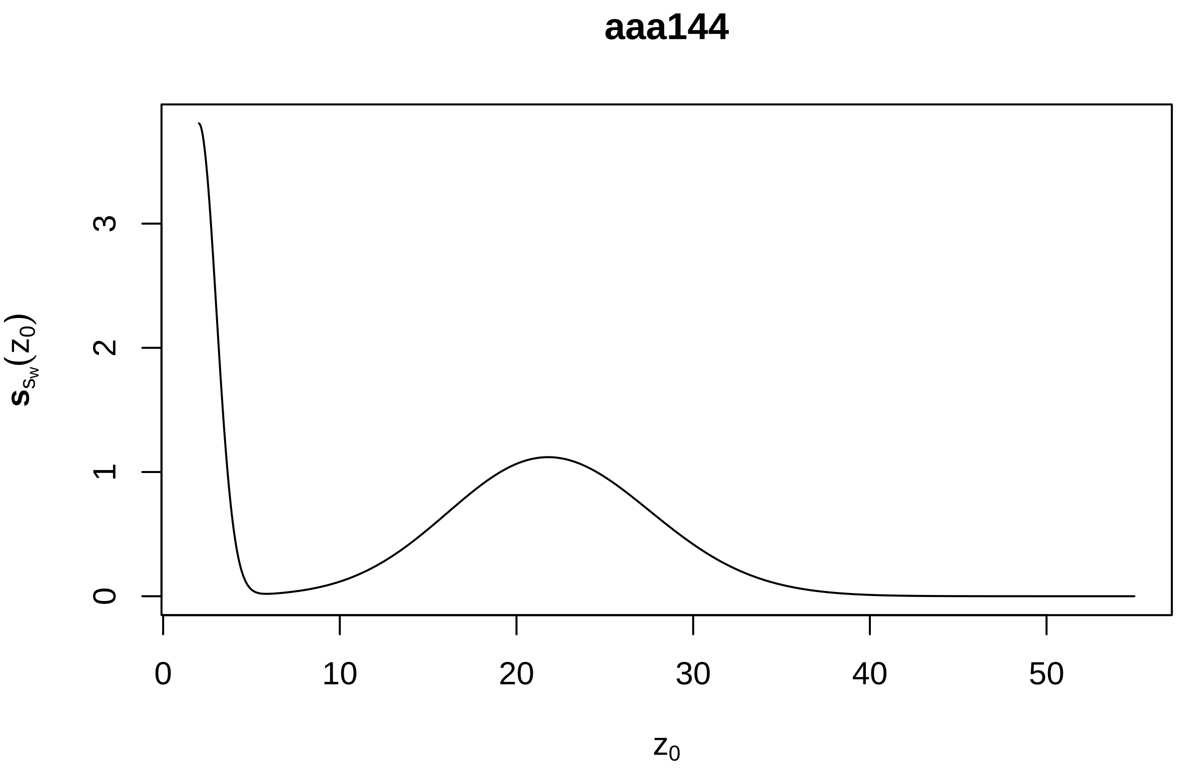


Other perturbation kernels

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 $\mathbf{s}_s(z_0) = \frac{(v s_s G) \circ w}{\langle v, w \rangle}$. We'll drop the first model from our lists because this analysis doesn't 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]])
```

```r
d_z <- d_zs[[3]]

numerator <- left_mult(s_s * G, v) * 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]))
```



Mean size at death and size at death kernels

To write...

Citations

1. Caswell, H. (2001) Matrix population models: construction, analysis, and interpretation, 2nd edn. Sunderland, MA: Sinauer Associates Inc
2. Wickham, H., François, R., Henry, L., & Müller, K. (2021). dplyr: A Grammar of Data Manipulation. R package version 1.0.6. https://CRAN.R-project.org/package=dplyr
3. Shelley, M. (1818) Frankenstein; or, the Modern Prometheus. London, Lackington, Hughes, Harding, Mayor, & Jones.
4. 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