IPMpack: an R package for demographic modeling with Integral Projection Models (v.1.2)

Jessica Metcalf, Sean M. McMahon, Rob Salguero-Gomez, Eelke Jongejans

June 7, 2012

The goal of IPMpack is to provide a suite of demographic tools based on Integral Projection Models (IPMs) to support biologists interested in making projections for populations where demography is strongly linked to a continuous variable, such as size. The package includes functions that can take data, such as size or age, as well as environmental covariates, and build models of growth, survival and fecundity. Functions are defined that then take these statistical models and construct IPMs. IPMpack has tools that compare different functional forms for the underlying statistical models, plotting them and returning AIC scores, as well as tools for diagnostic tests of the IPM models themselves. There are also methods to build population models for varying environments, use Bayesian methods to sample population parameters, estimate longevity and passage time, sensitivity and elasticity (of either parameters or matrix elements), and much more.

This vignette is intended to introduce the biologists with a wide range of quantitative skills to the concepts of IPMs as well as the implementation of IPMpack. This vignette is for IPMpack version 1.2, and so we encourage users to contact the IPMpack team at IPMpack@gmail.com with any feedback or mistakes they find. We also host a blog at R-forge IPMpack Web Site that contains news of updates, new features, and announcements of papers and meetings relevant to IPMs.

1 Introduction to Integral Projection Models

An Integral Projection Model (IPM) is a demographic tool to explore the dynamics of populations where individuals' fates depend on state variables that are continuous (e.g., weight, diameter at breast height, height, limb length, rosette diameter) or quasi-continuous (e.g., number of leaves, age, number of reproductive structures) and may be a mixture of discrete and continuous. IPMs track the distribution of individuals n across these state variables between census times (e.g., year t and year $t+1$) by projecting from models that define the underlying vital rates (e.g., survival, growth, and reproduction) as a function of the (quasi-)continuous state variables. For detailed introductions to IPMs, see Easterling et al. (2000), and Ellner & Rees (2006, 2007).

Briefly, an IPM is defined by a kernel K that represents probabilities of growth between discrete or continuous stages, survival across these stages, and the production of offspring and offspring recruitment. For example, in the simplest case, where the population is structured by a continuous covariate, size, then

$$n(y,t+1) = \int_L^U K(y,x)n(x,t)\,dx \tag{1}$$

where $n(y,t+1)$ is the distribution across size y of both established and new individuals in census time $t+1$, $n(x,t)$ the distribution across size of individuals in census time t, and L and U the lower and upper size limits modeled in the IPM, respectively.

Multiple functional forms for both demographic processes as well as their error structures can be easily accommodated with IPMpack. The F kernel (equation 4) describes per-capita contributions of reproductive individuals to number of new individuals at the next census. Multiple size-dependent or size-independent vital rates can be fitted within the F kernel, reflecting for example reproductive probability, number of reproductive structures (e.g. flowers in plants, basidia in fungi), number of propagules within reproductive structure (e.g. seeds for plants), and so on. Additionally, a range of constants (c_1, c_2, ...) can be included if there are no data for a stage. Finally, the F kernel definition includes a probability density function describing the size of offspring recruiting into the population, f_d,

$$n(y,t+1) = \int_L^U K(y,x)n(x,t)dx = \int_L^U [T(y,)+F(y,x)]n(x,t)dx \tag{2}$$

where

$$\int_L^U T(y,x)n(x,t)dx = \int_L^U surv(x)growth(y,x)dx \tag{3}$$

$$\int_L^U F(y,x)n(x,t)dx = \int_L^U c_1c_2c_3...fec1(x)fec2(x)fec3(x)...f_d(y,x)dx \tag{4}$$

After numerically solving these kernels, key ecological and evolutionary quantities such as the population rate of increase λ, the stable population size structure, the net reproductive rate R_0, and many others can be estimated (see Caswell 2001 for more a comprehensive discussion).

Essentially, the same tools are available for IPMs as for discrete projection matrices (matrix population models), e.g., estimation of population growth rate, sensitivities, elasticities, life table response experiment [LTRE] analyses, passage time calculations, etc (Caswell 2001, Cochran & Ellner 1992, and others). The main difference between an IPM and a matrix model is that while in discrete projection matrices the number of classes (i.e., number of stages in the life cycle of the study species) must be defined `a priori`, IPMs impose the discretization of the three-dimensional surface defined by equation 1 in the last step. This produces a typically large matrix (e.g., 100 x 100 cells) that is more robust to biases from matrix dimensionality (Zuidema et al. 2010, Salguero-Gomez & Plotkin 2010) and sample size (Ramula et al. 2009) than classical matrix models.

The goal of IPMpack is to provide a centralized set of quantitative techniques based on IPMs to help ecologists and evolutionary biologists model populations. IPMpack v. 1.2 can accommodate multiple vital rates from complex life cycles all grouped into two main sub-kernels: T and F (equation 2) [1].

This vignette will now walk through the steps of a basic IPM analysis. We first describe the kind of data necessary to build an IPM. If a user begins 'from scratch', they must input data in a specific format (described below). However it is possible to jump past this step and use IPMpack capabilities on IPMs that were developed outside of IPMpack. That is, if a user wants quick diagnostic routines, figures and summary statistics on an IPM matrix already built, IPMpack can readily accommodate that. However there are some features that, because of the object-oriented coding require some specific structures (and other features that do not). Please refer to the manual files and the rest of this vignette for this information. But however a user wants to implement IPMpack, the vignette will begin at the beginning with data set up. We will then walk through how to build and analyse a basic IPM model. More complex models will be introduced later, with options to create unique class objects and methods, as well as run comparative model testing and Bayesian implementations.

2 Getting started: setting up the data for IPMpack

For users who prefer to define IPM matrices using their own statistical tools, there is no requirement for the data to be in any particular format, and most of the functions in IPMpack will operate on the matrices directly (e.g., life expectancy, sensitivity of matrix elements, etc). However, to use IPMpack's full capacities, the individual-level demographic data must be organized in a specific format in R: a `data frame` where each row represents one observation of an organism in the population at one census time t with the following column names:

- `size`: size of individuals in census time t *
- `sizeNext`: size of individuals in census time $t+1$ *
- `surv`: survival of individuals from census time t to $t+1$ (contains: 0 for death or 1 for survival) *
- `fec1, ...`: as many columns as desired relating size to sexual reproduction. For example, this might be:
  - `fec1`: probability of reproduction (output: 0 for no reproductive or 1 for reproductive)
  - `fec2`: number of reproductive structures (output: 1, 2, 3, ...) when individual is reproductive, that is, when fec1 = 1
  - `fec3`: number of propagules (output: 1, 2, 3, ...) per reproductive structure (e.g. seeds per flower in reproductive plant individual)
  - ...

  The default construction for the analytical part of IPMpack is such that any columns for which the column label contains "fec" will be included in the analysis of the reproductive part of the life cycle (kernel F) automatically. This default can be over-ridden so that specific columns are identified for IPMpack functions to use.
- `stage`: stage of individuals in census time t. For rows in the data frame where `size` is not an NA, then this must be the word "continuous". Where `size` is NA, any variety of named discrete stages may be defined (e.g. "seed bank"). If this column is missing, many procedures

[1] Note than in the seminal paper by Easterling et al. (2000) this kernel was referred to as P, but here we follow the terminology by Caswell (2001) and call it T instead). The T kernel (equation 3) describes `growth` between demographic censuses conditional on individuals' survival (`surv`).

in IPMpack are designed to simply fill in this column assuming that only "continuous" state variables describe the life cycle of the species, i.e. there are no discrete stages. For running `makeFecObj`, the column must be a factor. If not supplied, the function will generate this column assuming all individuals are "continuous".

- `stageNext`: stage of individuals in census time $t+1$, in the simples case, "continuous" or "dead". As above, this column is not essential for many procedures in IPMpack. For running `makeFecObj`, the column must be a factor. If not supplied, the function will generate this column assuming all individuals that are alive are "continuous".
- `number`: number of individuals corresponding to each row in the data frame. For all rows corresponding to movement between continuous stages, this value will be 1, but for movement between discrete stages (e.g., from "dormant seeds" to "seeds ready to germinate") then this number may be > 1, potentially directly reflecting observed individuals in the data. This information avoids having a data frame with a row for every discrete stage (e.g. seed). As above, many proceedures in IPMpack will simply assume that this value is always 1.
- `covariate`: value of a discrete covariate in census time t, such as light environment at time t, age at t, patch at t, etc.
- `covariateNext`: value of a discrete covariate in census time $t+1$.
- ...any other covariates of interest, named as desired by the user are possible too (e.g., precipitation, habitat, temperature, etc).
- `offspringNext`: if the size contained in sizeNext corresponds to the size of an offspring, this column will contain either the value "sexual" or "clonal" (depending on whether sexual or clonal reproduction is being considered). If this column exists, rows that take these two values will be excluded from the growth analyses (functions `makeGrowthObj` and variants thereof, see below).

The * symbol above indicates the minimum columns in the data frame required to obtain passage time and life expectancy calculations. These values form the T kernel. If sufficient additional columns are available, a full life-cycle model, containing the F kernel, can be produced and further analyses are possible. Although `size` and `sizeNext` can be transformed, many of the utility functions assume no transformations in columns in the original data frame pertaining to fertility. Transformations can be formally called in various parts of the package and appropriate F matrices built that account for these transformations. In addition, users may also define IPMs independently, and then introduce them into IPMpack for application of further utility functions (sensitivities, stochastic growth rates, etc).

3 The basics: building an IPM

First, the user must load the IPMpack package from cran into R.

```
> library("IPMpack")
```

Next, the user must input demographic data. As mentioned above, most functions of IPMpack require a data file with at minimum columns called `size`, `sizeNext`, `surv`, where 'size' is size at time t, 'sizeNext' is size one census later, and 'surv' is a series of 0s and 1s, indicating if the individual survived or not. In the case of 'size' and 'sizeNext', data can be transformed (e.g., onto a log scale), if appropriate via functions built into IPMpack. For the purpose of learning how to use IPMpack, the user can either use his/her own data (adjusted to have the appropriate headings, as aforementioned), or generate them with a function built into IPMpack:

```
> dff <- generateData()
```

A quick check indicates that this contains sensible (fictional) information:

```
> head(dff)
```

```
      size sizeNext surv covariate covariateNext        fec      stage
1 3.574855 2.673455    0         1             0  0.0000000 continuous
2 6.689564 6.422653    1         0             0 21.0993092 continuous
3 5.445966 4.258094    1         1             1  0.0000000 continuous
4 5.159955 4.109065    0         1             0 13.0878897 continuous
5 6.158777 5.394140    1         1             0  0.0000000 continuous
6 5.543445 5.517133    1         1             0  0.2928447 continuous
   stageNext
1 continuous
2 continuous
3 continuous
4 continuous
5 continuous
6 continuous
```

for simplicity, no discrete covariates are included in this first example. Figure 1 (p. 6) is produced by the following code:

```
> plot(dff$size, dff$sizeNext, xlab = "Size at t", ylab = "Size at t+1")
```

IPMpack is written in object-oriented code, using *R S*4 objects. This means that extra object classes are used by IPMpack, with methods assigned to those classes that do particular things to specific objects. An example for those familiar with *R* is the `plot` function. When applied to two vectors, it produces an x-y plot, but when applied to a fitted linear regression, it provides a series of diagnostic plots. In other words, the 'plot' method is object-specific and does different things to objects of class 'numeric' and objects of class 'lm'.

IPMpack contains defined classes for growth, survival and fertility objects, and associated methods that allow the user to build IPM objects. In addition, this object-oriented structure in IPMpack uses methods from IPM objects to calculate life expectancy, passage times, and other population estimates of interest. The advantage of object-oriented programming is its flexibility: for example, the same machinery can be applied to suites of underlying regression forms and the user can take advantage of pre-existing highly generalized *R* functions, such as `predict`. The needs any particular dataset may require different object and method definitions. Towards the end of this vignette we also describe how to define a new class and a new method (e.g., a new growth object for a specific life-history structure, and a new growth method applicable to plotting infomration from that object).

As an example, let us first define objects built as simple polynomial regressions from the generated data. The source code of `generateData` will confirm that the survival data is built around a polynomial logistic regression relating size at t to survival from t to $t+1$, and the growth data is built around a polynomial regression relating size at t to size at $t+1$. To make growth and survival objects that reflect this, the user must implement:

```
> gr1 <- makeGrowthObj(dataf = dff, explanatoryVariables = "size+size2",
+                      responseType = "sizeNext")
> sv1 <- makeSurvObj(dff, explanatoryVariables = "size+size2")
```

In both these functions, the argument `explanatoryVariables` contains formulas of the type used in linear or logistic regressions in *R*, built around the possible defined range of transforms of `size` currently available (`size2` which is size^2, `size3` which is size^3, and `logsize` which is log(size). Currently further transforms of `size` are not possible. This function can also be used to fit models that include a single discrete covariate (e.g., light environment, age, etc) as long as this exists in the `dataf` in a column named `covariate`. For instance, the user could model the population

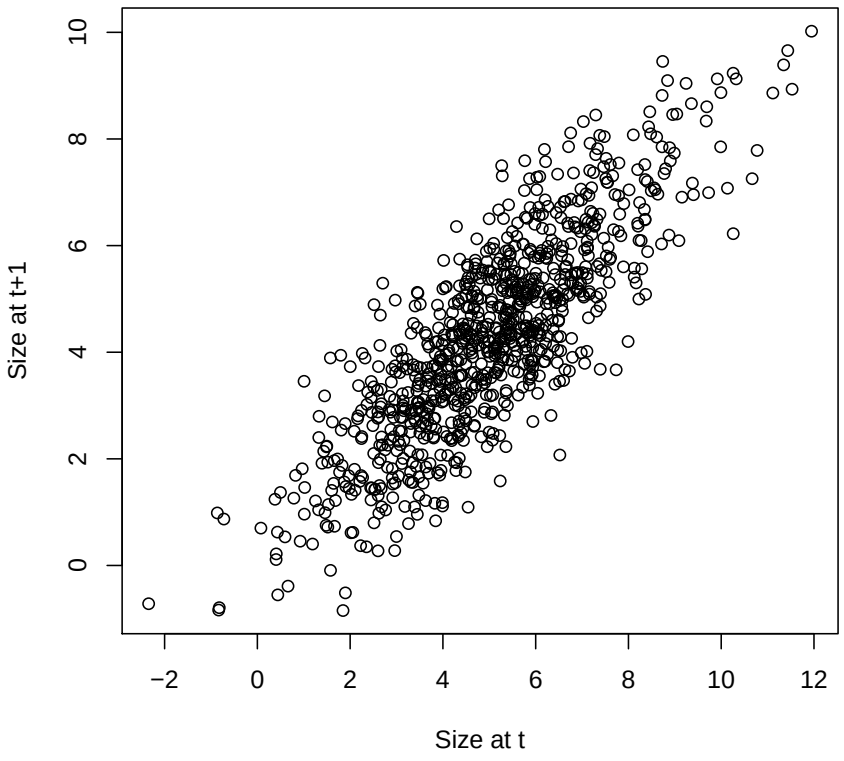

Figure 1: Size at t and size at t+1

dynamics according to `size + covariate` or `size + logsize*covariate`, etc. For the growth model, possibilities for `responseType` are: `sizeNext` meaning that the reponse variable is size at the next census time, or `incr` meaning that the response variable is the size increment that has accrued between the two census times (common among tree demographic studies), and `logincr` meaning that the response variable is the log of the size increment that has acrrued between the two census intervals.

Below, the functions `makeGrowthObjManyCov` and `makeSurvObjManyCov` are introduced, which allow any covariates that exist in `dataf` to be fitted (e.g., `size + temperature + site`, etc) via the argument `explanatoryVariables`. The functions are different from the above, since in this case, a slightly different type of growth and survival object needs to be defined to allow slightly different growth and survival methods to be applied.

Glancing at the source code will confirm that all these functions simply fit a linear regression relating size at t+1 or increment to size at t and covariates for growth, as for survival. The survival and growth objects created have a slot called 'fit' that holds the regression.

```
> gr1

An object of class "growthObj"
Slot "fit":

Call:
lm(formula = Formula, data = dataf)

Coefficients:
(Intercept)         size        size2
  0.3999816    0.7706199    0.0009759


Slot "sd":
[1] 1.057481
```

IPMpack contains two functions that allow the user to check these two relationships against the data used for them in order to explore goodness of fit and effect of mesh size, shown in Figure 2 (p. 8).

```
> par(mfrow = c(1, 2), bty = "l", pty = "m")
> p1 <- picGrow(dff, gr1)
> p2 <- picSurv(dff, sv1, ncuts = 30)
```

To build a demographic model describing survival and growth transitions from these objects, the user can use the function `createIPMTmatrix`, i.e.:

```
> Tmatrix <- createIPMTmatrix(nBigMatrix = 50,
+                             minSize = -5, maxSize = 35,
+                             growObj = gr1, survObj = sv1,
+                             correction = "constant")
```

where `nBigMatrix` is the number of bins used, `minSize` and `maxSize` define the limits of the IPM, U and L in the equations above. Typically, these range should usually extend to beyond the smallest and largest size measurement, but the user might want to exclude outliers). The objects growObj and survObj define changes in size and survival as defined above. IPMpack includes an useful function `diagnosticsTmatrix` that provides a series of plots indicative of whether bin choice and size range is adequate. Applying this function as a preliminary step before obtaining demographic and evolutionary output from IPMs is highly recommended at this stage (see `?diagnosticsTmatrix` for details). The argument `correction = "constant"` will rectify some of the more egregious numerical slippage in the model defined above, but it will do this in a slightly

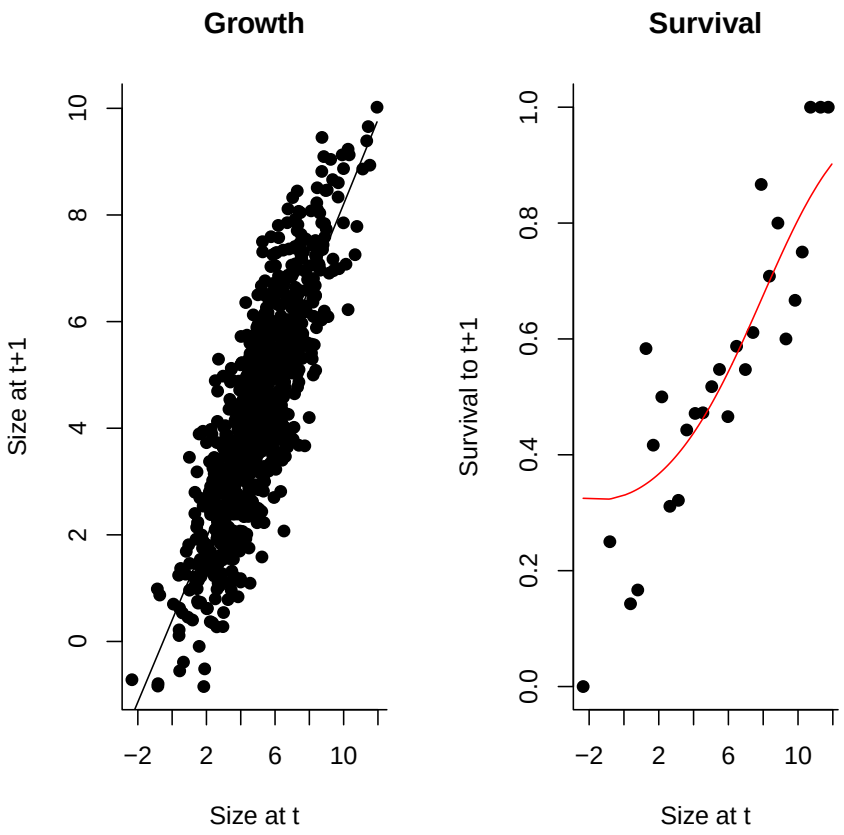

Figure 2: Growth and survival objects

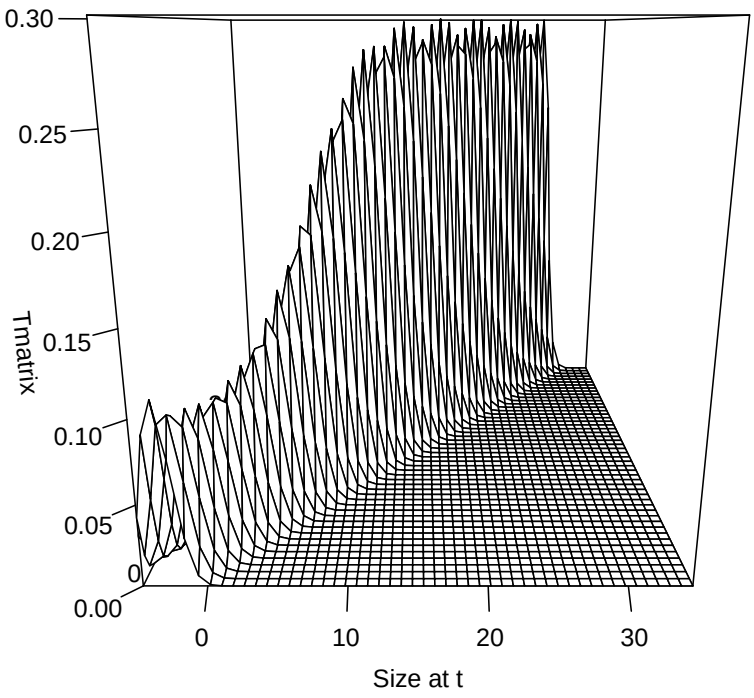

Figure 3: Transition matrix encompassing survival and growth transitions only

arbitrary way (i.e. simply adding a constant value to all elements of each column in the matrix, which may or may not be appropriate), so it is worth exploring options in detail.

The `createIPMTmatrix` function builds around methods defined so that it will provide appropriate output whatever the survival and growth objects are (e.g. error structure, covariates...). The T matrix contains a matrix defining the transitions, but also other useful slots, e.g., the meshpoints, etc. The user can access this information by writing:

```
> slotNames(Tmatrix)

[1] ".Data"          "nDiscrete"      "nEnvClass"      "nBigMatrix"
[5] "meshpoints"     "env.index"      "names.discrete"
```

and finally, the user can plot the Tmatrix using `persp` (Figure 3). Next, with this, the user can obtain the life expectancy, and passage time to a chosen size (here set at the mean) for the range of meshpoints

```
> LE <- meanLifeExpect(Tmatrix)
> pTime <- passageTime(mean(dff$size, na.rm = TRUE), Tmatrix)
```

and the user can also plot these againts `Tmatrix@meshpoints` to examine how life expectancy and passage vary as a function of size (Figure 4 p. 10). The function `run.Simple.Model` takes as minimum arguments a data frame and a target size (i.e., here type: `runSimpleModel(dff, chosenSize = 4)`) and runs this analysis to create figures for survival, growth, life expectancy

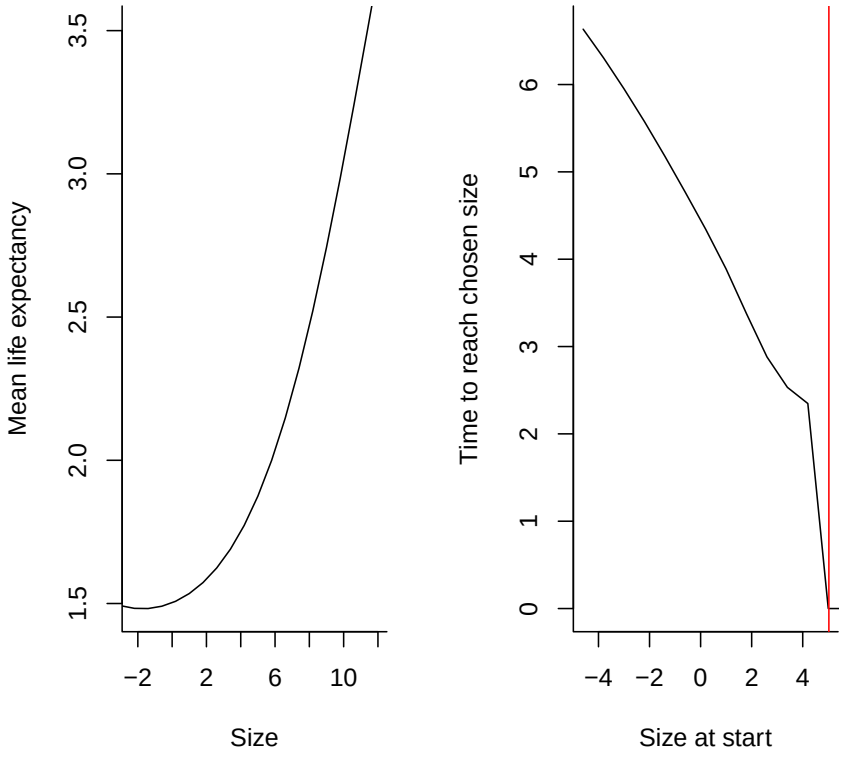

Figure 4: Associated Life Expectancy and Passage Time

and passage time as shown so far, assuming the simplest possible models of survival and growth (basic linear and logistic regressions, no covariates, etc).

If the user defines a fertility object -which for instance is not always easy with for example trees- IPMpack can also create a transition matrix describing movement between sizes attributable to fertility.

```
> fv1 <- makeFecObj(dff, explanatoryVariables = "size",
+                   Family = "gaussian",
+                   Transform = "log")
> Fmatrix <- createIPMFmatrix(nBigMatrix = 50, minSize = -5,
+                             maxSize = 35,
+                             fecObj = fv1,
+                             correction = "constant")
```

Note that `makeFecObj` can either just scan the `dataf` and extract all the columns that contain the letters "fec" (the default, as explained above) and fit them in alphabetical sequence using the predictors defined in `explanatoryVariables` and using the family defined in `Family` with transforms defined in `Transform` (in the alphabetical sequence); or `fecNames` can be defined in an argument to `makeFecObj`, and this tells *R* which columns to select and fit fertility predictors to as in the previous, where `Family`, `Transform` etc, will be applied in the order `fecNames`. Please note that the fecundity columns must not be transformed in the data frame if `makeFecObj` function is used since IPMpack will perform appropriate transformations in the fitting according to the argument `Transform` and will use these appropriately in functions designed to build the F matrix.

The default arguments required to run `makeFecObj` to create a fecundity object from which an F matrix with no discrete stage can be built are `offspringSplitter=data.frame(continuous=1)`, `offspringTypeRates=data.frame(NA)` and `fecByDiscrete=data.frame(NA)`. Additionally, note that if there are values other than `"continuous"` in the `stage` column of the data-frame named `dff` in the example above, then the function will assume that multiple offspring classes are required, and the result will be an IPM with nBigMatrix + the number of offspring classes deduced (which is the number of names in `stage` other than `"continuous"`). This may lead to a mismatch with the size of the T matrix unless a discrete transition matrix is explicitly being included in the T matrix (see below, incorporating discrete stages).

If the data-frame contains an extra column `offspringNext` that takes the values `sexual`, and that corresponds to rows where both `size| and sizeNext are different from NA, the user can define a relationship between maternal size and offspring size through the makeFecObj argument offspringSizeExplanatoryVariables. The default is to only fit an intercept, equivalent to simply having a mean and variance of offspring size. The function makeFecObj also allows users to simply over-write the mean and variance of offpsring size with the values of their choice (arguments meanOffspringSize and sdOffspringSize).`

`The function makeClonalObj operates identically to makeFecObj except that it looks for columns that contain "clon" in them as a default for fitting fertility relationships too, and offspring are only considered for fitting the distribution of mean and standard deviation of offspring size if the column offspringNext takes the values clonal.`

`The user can combine the F matrix with (an identically built, i.e., same bin number, size limits and discrete classes) survival-growth transition T matrix to obtain a full Integral Projection Model, and its population growth rate` λ`, sensitivity, elasticity, etc.`

```
> IPM <- Tmatrix + Fmatrix
> eigen(IPM)$value[1]

[1] 2.453149

> sensitivity <- sens(IPM)
> elasticity <- elas(IPM)
```

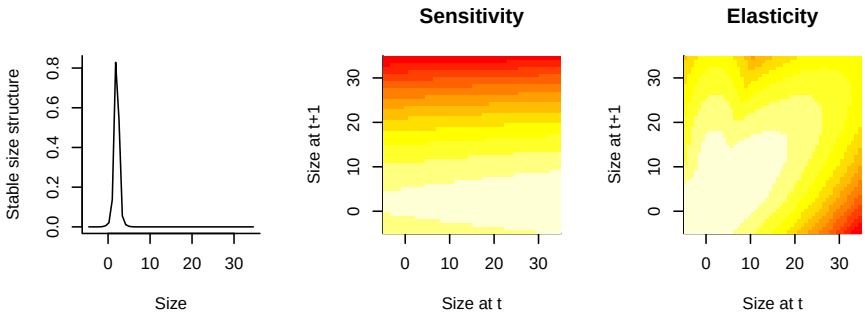

Figure 5: Measures off a full IPM

These outputs can be plotted against the meshpoints (Figure 5 p. 12). In addition to perturbation measures from mesh cells, the user can also obtain sensitivity and elasticity of particular parameters that underlie the kernels, e.g., doing:

```
> res <- sensParams(growObj = gr1, survObj = sv1, fecObj = fv1,
+                   nBigMatrix = 50, minSize = -5, maxSize = 15)
> res

$slam
    grow (Intercept)            grow size           grow size2
           0.12379137           0.27127038           0.65002017
            sd growth     surv (Intercept)            surv size
           0.04342693           0.24047997           0.52105802
           surv size2 reprod 1 (Intercept)        reprod 1 size
           1.23319039           2.44146919           4.58676640

$elam
    grow (Intercept)            grow size           grow size2
        0.0495167451         0.2090568129         0.0006343539
            sd growth     surv (Intercept)            surv size
        0.0458746622        -0.1702719543         0.0252755405
           surv size2 reprod 1 (Intercept)        reprod 1 size
```

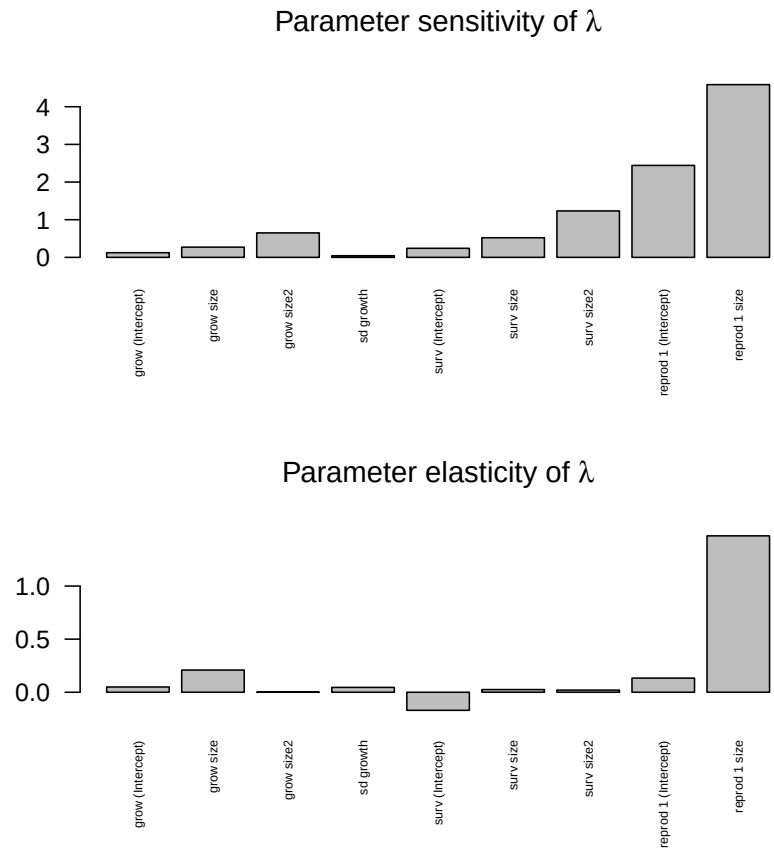

Figure 6: Sensitivity and elasticity of parameter values

```
       0.0202776202          0.1332205758          1.4728797642
```

and this output can be plotted out (Figure 6 p. 13) using

```
> par(mfrow = c(2, 1), bty = "l", pty = "m")
> barplot(res$slam, main = expression("Parameter sensitivity of "*lambda),
+                     las = 2, cex.names = 0.5)
> barplot(res$elam, main = expression("Parameter elasticity of "*lambda),
+                     las = 2, cex.names = 0.5)
```

4 Discretely varying environments

A first possible extension of IPMs is to create a compound IPM matrix where, in addition to moving between continuous sizes, individuals move through discrete environments where the discrete environmental states have an expected sequence, and therefore can be described by a transition matrix of their own (e.g. light environments for tropical trees, as in Metcalf et al. 2009).

To explore this type of dynamics, the user needs to either provide or simulate an environmental variable at t and the corresponding value at $t+1$. Here, it has been generated as part of the generateData function (See above). From this generated data, the user can then create an environmental transition matrix, which describes how the

environment tends to move between these states from one census time to the next. If the data has been set up as described, there is a function that will do this for the user:

```
> env1 <- makeEnvObj(dff)
> env1

An object of class "envMatrix"
          [,1]      [,2]
[1,] 0.1926952 0.1796117
[2,] 0.8073048 0.8203883
Slot "nEnvClass":
[1] 2
```

The user can now use IPMpack to create a survival-growth transition T matrix that encompasses movement across environments, first redefining the survival and growth objects to fit a discrete covariate, by changing the explanatoryVariables argument:

```
> gr1 <- makeGrowthObj(dff, explanatoryVariables = "size+covariate")
> sv1 <- makeSurvObj(dff, explanatoryVariables = "size+covariate")
```

Note that these functions will only work appropriately for a discrete covariate if the value of the covariate at time t is available as a column in the data frame names covariate and the value of the covariate at the next census is available as a column in the data frame called covariateNext. IPMpack functions use the presence of a column in the data frame called covariate as a cue to renumber values in these two columns to numeric levels between 1 and the observed number of covariate levels to facilitate looping, and changes them into factors. Once this step is implemented, the user can use these functions to create a compound T matrix, using createCompoundTmatrix:

```
> Tmatrix <- createCompoundTmatrix(nBigMatrix = 50, minSize = -5,
+                                  maxSize = 35,
+                                  envMatrix = env1, growObj = gr1,
+                                  survObj = sv1,
+                                  correction = "constant")
```

Essentially, the compound T matrix is a large matrix with stacked IPMs corresponding to each environment, modified to reflect movement between environmental states defined by env1. Passage time can be calculated using similar function, but now including the environmental matrix as an argument (equivalent life expectancy functions are in development):

```
> pTimes <- stochPassageTime(Tmatrix@meshpoints[15], Tmatrix, env1)
```

The resulting vectors contain the life expectancy and time to reach each size for individuals starting in each different environmental class, concatenated together (i.e. there are nBigMatrix values in the LE matrix ranging over the first environment, then nBigMatrix values ranging over the second environment, etc). The user can plot these against meshpoints (Figure 7 p. 15), each colour indicates a different starting environment. Similar syntax can be used for passage time (although note that here the function name has changed).

(Figure 7) :

With a fertility object, the user can also define a full life cycle IPM model for this environment. With such an information, obtaining the stochastic population rate of increase λ_s in this environment is relatively straight-forward. IPMpack does this

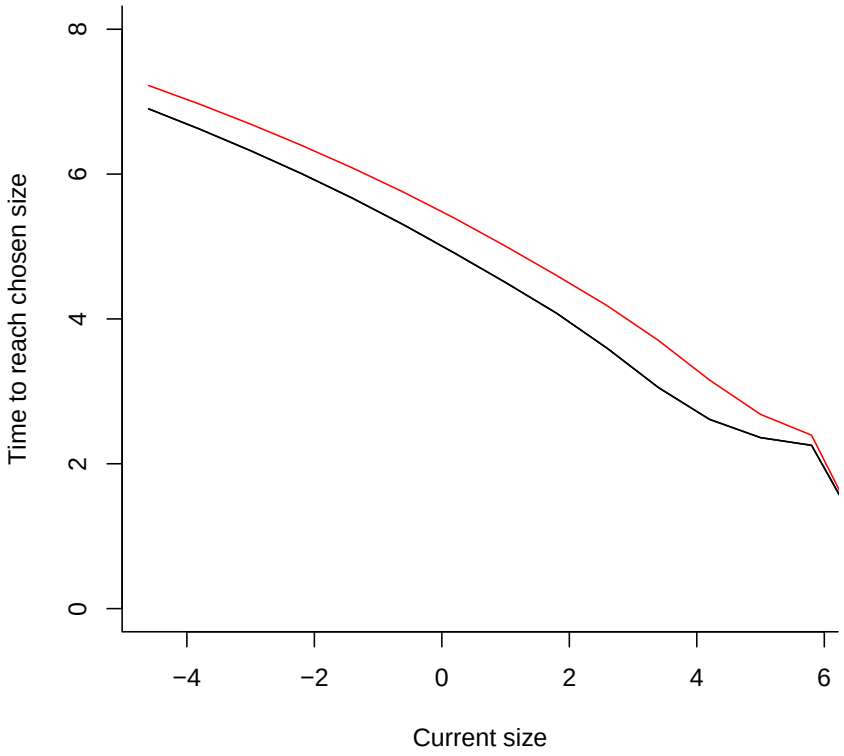

Figure 7: Passage time for a compound IPM; different colours reflect predictions for individuals starting in different environments

by sampling a very large number of environments and corresponding IPMs, and multiplying them together (Childs et al. 2004). At the moment, this is only defined for the case where environments (defined by the discrete covariates) are distributed independently (i.e. the next state does not depend on the previous state). To do this, the user must first define a list of IPMs (each the sum of a matrix of survival-growth transitions, and a matrix of fecundity transitions corresponding to a particular environment):

```
> IPMlist <- makeListIPMs(dataf = dff, nBigMatrix = 25, minSize = -5,
+                         maxSize = 35, explSurv = "size+covariate",
+                         explGrow = "size+size2+covariate",
+                         explFec = "size", Transform="log",correction = "constant")
```

Note that in this example IPMpack uses an arbitrary selection of explanatory variables for all the various linear and logistic regressions (explGrow, explSurv, etc). In reality, careful model selection will be used to establish this. Additionally, the number of environment types should in principle be greater than the two or three used here. Next, the user can estimate λ_s using:

```
> stochGrowthRateSampleList(listIPMmatrix = IPMlist,
+                           nRunIn = 30, tMax = 50)
```

```
[1] 0.9286943
```

where nRunIn defines the number of time steps to discard from the start of the time series in order to remove transient dynamics, and tMax is the total number of time steps to run, and should be large enough that increasing it does not substantially change the result (numbers presented here for efficiency are almost certaintly not large enough).

5 More generally varying environments

An alternative way of inhabiting stochastic environments is to experience continuously changing covariates (rather than moving between discrete states, as the above describes). In this case, rather than building a single megamatrix, desired variables are obtained by multiplying up a suite of matrices and relying on the weak ergodic theorem for convergence (as described for obtaining λ_s, above). IPMpack contains code to do this. The user must first define a new data frame containing several time-varying covariates, and then, build the associated survival, growth and fertility objects:

```
> dff <- generateDataStoch()
> sv1 <- makeSurvObjManyCov(dataf = dff,
+                           explanatoryVariables = "size+covariate1+covariate3")
> gr1 <- makeGrowthObjManyCov(dataf = dff,
+                             explanatoryVariables = "size+covariate1+covariate2")
> fv1 <- makeFecObj(dataf = dff, fecConstants = data.frame(1.8),
+                   explanatoryVariables = "size", Transform = "log")
```

As before, the user can explore the data:

```
> head(dff)
```

```
      size  sizeNext surv covariate1  covariate2 covariate3       fec
1 8.168331 11.760154    1  1.1996198 -0.01175445 -0.5007984 14.6829535
2 5.879738  8.529391    0  0.7915191 -0.23756469 -0.5793250 16.2174887
```

```
3 2.244922  0.989347    0 -0.6275048 -1.27836017 -0.6793564  0.0000000
4 3.751224  1.237328    1 -1.0780716  1.44193818  0.8319685  0.9878364
5 4.619693  4.339719    1 -0.2751880  0.81041941 -0.2364020 17.6166299
6       NA -2.129319    1  0.4549957  1.06008520 -0.2962469  0.0000000
       stage  stageNext number
1 continuous continuous      1
2 continuous continuous      1
3 continuous continuous      1
4 continuous continuous      1
5 continuous continuous      1
6       <NA> continuous      1
```

and glance at the objects, e.g.,

```
> gr1

An object of class "growthObjMultiCov"
Slot "fit":

Call:
lm(formula = Formula, data = dataf)

Coefficients:
(Intercept)         size   covariate1   covariate2
   1.022484     0.897668     3.005807     0.008345


Slot "sd":
[1] 0.2173566
```

From these data, to explore predicted demographic outcomes for the model, the user must decide on a time scale and length for investigation, and define it by a vector called 'tVals', here set to reflect monthly intervals over 4 years, with years as the time scale. With this, the user can then generate a time series that should look like the time series observed in the data. In the example below, covariates that vary seasonally were simulated, i.e., they fluctuates randomly around a sine wave which peaks once a year ('covTest'), and from this generate a matrix containing time as rows, and different covariates in columns.

```
> tVals <- seq(1, 4, by = 1/12)
> covTest <- c(1 + 0.5*sin(2*pi*tVals))
> covMatTest <- data.frame(covariate1 = rnorm(length(covTest), covTest, 0.5) - 1,
+                          covariate2 = rnorm(length(covTest), covTest, 0.5) - 1,
+                          covariate3 = rnorm(length(covTest), covTest, 0.5) - 1)
```

Note that if there is no apparent temporal pattern to the data one could simply generate random normal distributions of the covariates using their observed mean and variance. Other types of temporal patterns (multiannual, etc) are also possible. With this setup, the user can then estimate the stochastic growth rate over these years, using the geometric mean of the population growth rate (Tuljapurkar 1990; Childs et al. 2004), for these particular covariates using:

```
> r <- stochGrowthRateManyCov(covariate = covMatTest, nRunIn = 12*1,
+                             tMax = length(tVals), growthObj = gr1,
+                             survObj = sv1, fecObj = fv1, nBigMatrix = 20,
+                             minSize = 2*min(dff$size, na.rm = TRUE),
```

```
+                               maxSize = 1.5*max(dff$size, na.rm = TRUE),
+                               nMicrosites = 50, correction = "constant")
> print(r)
```

```
[1] 0.8445717
```

Setting nRunIn = $12*1$ in this example is equivalent to discarding the first 1 years (likely to contain transients) since the chosen time step is months. Note that in this formula, it was assumed that density-dependence acts on seedling establishment, and that 50 microsites are available for seedling establishment in every time step. Setting nMicrosites = 0 allows for calculations without density-dependence, and nMicrosites can also be a vector, if the number of microsites fluctuates through time. It may also be interesting to have a glance at what has been happening to the population structure over this time-course, and the function trackPopStructManyCov allows this; IPMpack also contains a dedicated function to depict the results from this, plotResultsStochStruct.

6 Incorporating discrete stages

Populations are often structured by both discrete and a continuous stages, for example, many plant populations may persist for many years in a seedbank as well as having size-determined fates after they germinate. IPMpack can incorporate this variability for complex life cycles (Ellner & Rees 2006). To illustrate this, the user must first generate data that includes both discrete and continuous life-history stages:

```
> dff <- generateDataDiscrete()
```

A quick check indicates that these data contain several types of stage classification (and not just "continuous" as seen up till now):

```
> table(dff$stage)
```

```
continuous    dormant   seedAge1    seedOld
       950         50         35         32
```

Given this data structure, the user can make a fertiliy object that reflects the fact that propagules (e.g., seeds) produced in one year may directly recruit into the continuous phase (e.g., seedling), or may end up in a discrete stage (e.g., seed bank). The makeFecObj (and similar functions) have an argument that allows the user to define this dichotomy, called offspringSplitter:

```
> fv1 <- makeFecObj(dataf = dff, Transform = "log",
+                   offspringSplitter = data.frame(continuous = 0.2,
+                   dormant = 0, seedAge1 = 0.8, seedOld = 0),
+                   fecByDiscrete = data.frame(dormant = 0,
+                   seedAge1 = 0, seedOld = 0))
```

In this example, 20 % of seeds produced at t end up in the continuous part of the population structure at $t+1$ (for example, they might directly recruit as rosettes from one year to the next) and 80 % of seeds recruit into the "one year old seeds" stage. Although in this case no individuals are recruited at $t+1$ into the "dormant" or "old seeds" stages (since these will come from adult plants or the seed bank), they are included as offspringSplitter is where IPMpack identifies all the existing discrete stages. The argument fecByDiscrete reflects the fact that none of the discrete classes addressed in this example are likely to directly produce offspring (which may not always be the case). The resulting fecundity object can be used with createIPMFmatrix in the usual way:

```
> Fmatrix <- createIPMFmatrix(fecObj = fv1, nBigMatrix = 5,
+                             minSize = min(dff$size, na.rm = TRUE),
+                             maxSize = max(dff$size, na.rm = TRUE),
+                             correction = "constant")
```

The user also needs a Tmatrix that reflects the same structure. The continuous part of the T matrix will be broadly the same as usual:

```
> gr1 <- makeGrowthObj(dataf = dff,
+                      explanatoryVariables = "size", responseType = "sizeNext")
> sv1 <- makeSurvObj(dff, explanatoryVariables = "size")
```

Movement in and out of discrete stages is defined via an add-on of a transition matrix, that is defined using:

```
> discTrans <- makeDiscreteTrans(dff)
```

which captures survival and transitions between discrete stages and the continuous stage (note that this function will not work unless the data frame dff contains appropriate columns stage and stageNext) and then the user can construct the T matrix using:

```
> Tmatrix <- createIPMTmatrix(nBigMatrix = 5,
+                             growObj = makeGrowthObj(dff),
+                             survObj = makeSurvObj(dff),
+                             discreteTrans = discTrans,
+                             correction = "constant")
```

Note that both the T matrix and the F matrix in this example have a rather small number of bins just for ease of comparison, and that a higher number is almost certainly advisable. The user can examine both matrices:

```
> print(Tmatrix)

An object of class "IPMmatrix"
              [,1]          [,2]          [,3]          [,4]          [,5]
[1,]  2.400000e-01 0.000000e+00 0.000000e+00  1.376925e-02  6.250908e-01
[2,]  0.000000e+00 0.000000e+00 0.000000e+00  0.000000e+00  0.000000e+00
[3,]  0.000000e+00 4.439560e-01 4.323308e-01  0.000000e+00  0.000000e+00
[4,]  7.599788e-01 7.252743e-02 1.127797e-01  4.509131e-01  1.393969e-20
[5,]  2.121858e-05 3.908116e-08 2.268288e-06  1.998321e-23  3.444674e-03
[6,]  2.448172e-25 6.012771e-25 3.309120e-19  5.196145e-91  4.994439e-31
[7,]  1.167289e-60 2.641338e-52 3.501656e-40 7.927588e-204 4.248827e-104
[8,] 2.299987e-111 3.312954e-90 2.687703e-69  0.000000e+00 2.120778e-222
              [,6]           [,7]          [,8]
[1,]  4.314031e-01  7.258043e-02  1.853533e-03
[2,]  0.000000e+00  0.000000e+00  0.000000e+00
[3,]  0.000000e+00  0.000000e+00  0.000000e+00
[4,]  7.186736e-80 1.759173e-172 2.684326e-294
[5,]  9.902597e-23  1.351601e-75 1.150000e-157
[6,]  8.005902e-11  6.093014e-24  2.890707e-66
[7,]  3.797652e-44  1.611609e-17  4.263381e-20
[8,] 1.056973e-122  2.501099e-56  3.689333e-19
Slot "nDiscrete":
[1] 3

Slot "nEnvClass":
```

```
[1] 1

Slot "nBigMatrix":
[1] 5

Slot "meshpoints":
[1]  4.1 14.3 24.5 34.7 44.9

Slot "env.index":
[1] 1 1 1 1 1

Slot "names.discrete":
[1] "dormant"  "seedAge1" "seedOld"
```

> print(Fmatrix)

```
An object of class "IPMmatrix"
     [,1] [,2] [,3]         [,4]         [,5]         [,6]        [,7]
[1,]    0    0    0 0.0000000000 0.0000000000 0.0000000000 0.000000000
[2,]    0    0    0 0.1393875095 0.2452378125 0.4314704014 0.759127254
[3,]    0    0    0 0.0000000000 0.0000000000 0.0000000000 0.000000000
[4,]    0    0    0 0.0032975319 0.0058016640 0.0102074239 0.017958900
[5,]    0    0    0 0.0131901724 0.0232067353 0.0408298349 0.071835844
[6,]    0    0    0 0.0140615045 0.0247397533 0.0435270205 0.076581261
[7,]    0    0    0 0.0039951487 0.0070290482 0.0123668787 0.021758236
[8,]    0    0    0 0.0003025199 0.0005322523 0.0009364425 0.001647573
            [,8]
[1,] 0.000000000
[2,] 1.335605378
[3,] 0.000000000
[4,] 0.031596815
[5,] 0.126387689
[6,] 0.134736757
[7,] 0.038281350
[8,] 0.002898733
Slot "nDiscrete":
[1] 3

Slot "nEnvClass":
[1] 1

Slot "nBigMatrix":
[1] 5

Slot "meshpoints":
[1] -0.505122  2.034272  4.573665  7.113059  9.652453

Slot "env.index":
[1] 1 1 1 1 1

Slot "names.discrete":
[1] "dormant"  "seedAge1" "seedOld"
```

and check for example that the slot namesDiscrete is aligned between them, and add them together:

```
> print(Tmatrix+Fmatrix)

              [,1]         [,2]         [,3]         [,4]         [,5]
[1,]  2.400000e-01 0.000000e+00 0.000000e+00 0.0137692513 0.6250908460
[2,]  0.000000e+00 0.000000e+00 0.000000e+00 0.1393875095 0.2452378125
[3,]  0.000000e+00 4.439560e-01 4.323308e-01 0.0000000000 0.0000000000
[4,]  7.599788e-01 7.252743e-02 1.127797e-01 0.4542106361 0.0058016640
[5,]  2.121858e-05 3.908116e-08 2.268288e-06 0.0131901724 0.0266514091
[6,]  2.448172e-25 6.012771e-25 3.309120e-19 0.0140615045 0.0247397533
[7,]  1.167289e-60 2.641338e-52 3.501656e-40 0.0039951487 0.0070290482
[8,] 2.299987e-111 3.312954e-90 2.687703e-69 0.0003025199 0.0005322523
             [,6]        [,7]        [,8]
[1,] 0.4314030911 0.072580433 0.001853533
[2,] 0.4314704014 0.759127254 1.335605378
[3,] 0.0000000000 0.000000000 0.000000000
[4,] 0.0102074239 0.017958900 0.031596815
[5,] 0.0408298349 0.071835844 0.126387689
[6,] 0.0435270206 0.076581261 0.134736757
[7,] 0.0123668787 0.021758236 0.038281350
[8,] 0.0009364425 0.001647573 0.002898733
```

The first three rows and columns concern transitions in and out of the discrete stages; the remainder are the usual T and F matrices describing moving across the continuous variables. The usual types of calculations (sensitivity via sens, life expectancy via meanLifeExpect, etc) can be applied here too.

7 Parameter uncertainty in a constant environment

First, the user must generate data again, and from them, a list of survival and growth objects reflecting the parameter posteriors of the fitted linear and logistic regression (taking the simplest case of structure only via a continuous covariate):

```
> dff <- generateData()
> grlist <- makePostGrowthObjs(dff,
+                              explanatoryVariables = "size",
+                              burnin=20,nitt = 40)
> svlist <- makePostSurvivalObjs(dff,
+                                explanatoryVariables = "size",
+                                burnin=20,nitt = 40)
```

Note that the data must not contain NAs. This function can also be used to set priors, etc. Note that the number of samples from the posterior used here nitt is rather small, and larger numbers are advisable. With output from this, the user can make lists of the T matrices:

```
> TmatrixList <- makeListTmatrix(grlist, svlist, nBigMatrix = 20,
+                                minSize = -5,
+                                maxSize = 35,
+                                correction = "constant")
```

If one of the lists is longer than the other, this function samples the shorter object at random to reach the size of the longer object. Note that in this example the matrix size is rather small just to save time, and larger number of bins are advisable. The function will also construct compound matrices, if an environmental matrix is provided. With this, the user can now obtain some posteriors for constant environment models.

```
> res <- getIPMoutput(TmatrixList, targetSize = 5, FmatrixList = NULL)
> names(res)

[1] "LE"          "pTime"       "lambda"      "stableStage"
```

The vector called λ and matrix called stableSize, etc, will consist of NAs, unless a list of Fmatrices is also provided, so that a complete population projection matrix can be built. IPMpack contains a similar function to obtain a list of F matrices, and if such a list is included as the third argument into the function getIPMOutput (for which the default is 'NULL'), the function will also return distributions of λ, the stable stage distribution, etc:

```
> fv <- makePostFecObjs(dff, explanatoryVariables = "size+size2", fecConstants=data.frame(0.01),
+                       burnin=20,nitt = 40, Transform = "log")

[1] 2

> FmatrixList <- makeListFmatrix(fv, nBigMatrix = 20, minSize = -5,
+                                maxSize = 35, cov = FALSE,
+                                correction = "constant")
> res <- getIPMoutput(TmatrixList, targetSize = 5, FmatrixList)
```

Again, larger number of iterations, binsize, etc, are recommended. The results can be visually inspected too (Figure 8 p. 23)

This is a rather slow way of proceeding - a large number of IPMs are being stored in R's memory. A slightly more rapid approach is to use the function getIPMOutputDirect that builds an IPM from a sample from the posterior, calculates relevant parameters, then over-writes this with a rebuilt IPM, etc.

8 Building your own objects and methods

If growth is best reflected by a saturating function, rather than the linear regression models provided, the user must define a new class of growth object:

```
> setClass("growthObjSaturate", representation(paras = "numeric", sd = "numeric"))
```

Then define the functional form of the mean prediction, with relevant parameters:

```
> fSaturate <- function(size, pars) {
+     u <- exp(pmin(pars[1] + pars[2] * size, 50))
+     u <- pars[3] * 1/(1+u)
+     return(u)
+ }
```

where the third parameter indicates the asymptotic size. The user can then estimate the parameters by fitting this function to the data using a wrapper function and optim.

```
> wrapSaturate <- function(par, dataf) {
+     pred <- fSaturate(dataf$size, par[1:3])
+     ss <- sum((pred - dataf$sizeNext)^2, na.rm = TRUE)
+     return(ss)
+     }
> tmp <- optim(c(1, 1, 1), wrapSaturate, dataf = dff, method = "Nelder-Mead")
> tmp
```

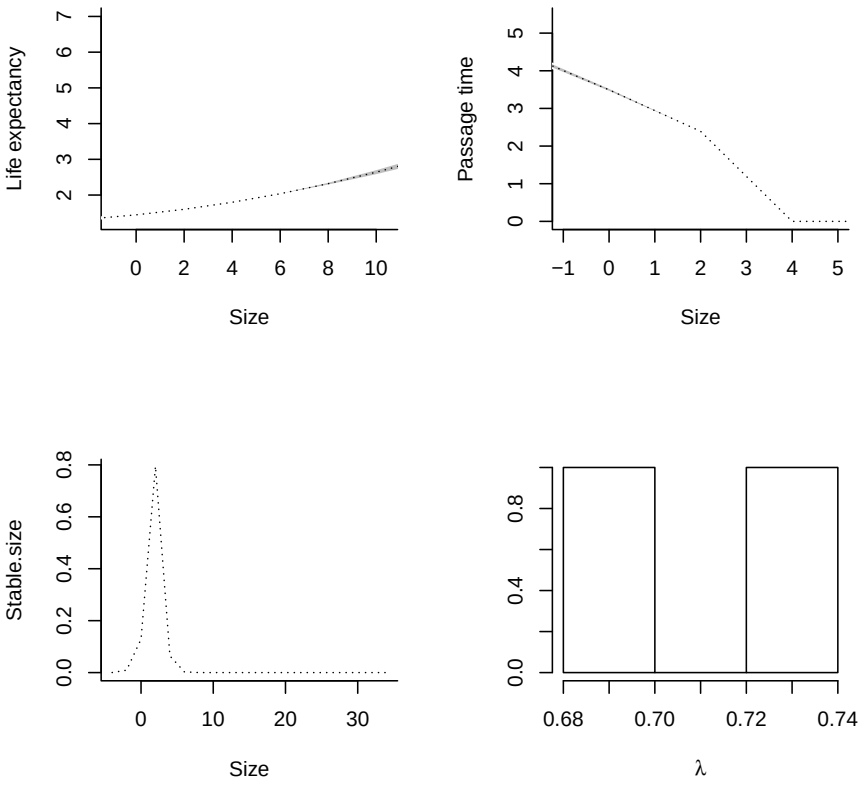

Figure 8: Uncertainty in IPM output

```
$par
[1]  2.140494 -0.354954 10.112288

$value
[1] 1079.024

$counts
function gradient
     286       NA

$convergence
[1] 0

$message
NULL
```

For simplicity, one can assume normally distributed errors:

```
> resids <- fSaturate(dff$size, tmp$par) - dff$sizeNext
> sdSaturate <- sd(resids, na.rm = TRUE)
```

With these parameters, the user can then define the new growth object:

```
> gr1 <- new("growthObjSaturate")
> gr1@paras <- tmp$par
> gr1@sd <- sdSaturate
```

Finally, the user must define a method appropriate for this type of object.

```
> setMethod("growth", c("numeric", "numeric", "numeric", "growthObjSaturate"),
+           function(size, sizeNext, cov, growthObj){
+               mux <- fSaturate(size, growthObj@paras)
+               sigmax <- growthObj@sd
+               u <- dnorm(sizeNext, mux, sigmax, log = F)
+               return(u);
+           })

[1] "growth"
```

By putting growthObjSaturate in the signature, R will use this particular method for all objects with this signature. Now, the user can go ahead and use all the other code as previously, without a need for further definitions.

If the user wishes to fit a growth model with, for example, gamma errors, a similar approach can be used, but with 'dgamma' instead of dnorm in the last line of growth method, and appropriate slots defined in the object, etc.

Selected References

- Caswell. 2001. Matrix population models: analysis, construction and interpretation. 2nd ed. Sinauer. Massachussetts, USA.
- Childs, Rees, Rose, Grubb & Ellner. 2004. Evolution of size-dependent flowering in a variable environment: Construction and analysis of a stochastic integral projection model. Proc. Roy. Soc. Lond. Ser. B. 271: 471-475.

- Cochran & Ellner. 1995. Simple methods for calculating age-based life history parameters for stage-structured populations. Ecological Monographs 62: 345-364.
- Ellner & Rees. 2006. Integral projection models for species with complex life-histories. American Naturalist 167: 410-428.
- Metcalf, Horvitz, Tuljapurkar & Clark. 2009. A time to grow and a time to die: a new way to analyze the dynamics of size, light, age and death of tropical trees. Ecology 90: 2766-2778.
- Rees & Rose. 2002. Evolution of flowering strategies in Oenothera glazioviana: an integral projection model approach. Proc. Roy. Soc. Lond. Ser. B. 269: 1509-1515.
- Ramula, Rees & Buckley. 2009. Integral projection models perform better for small demographic data sets than matrix population models: a case study of two perennial herbs. Journal of Applied Ecology 46: 1048-1053.
- Salguero-Gomez & Plotkin. 2010. Matrix dimensionality bias demographic inferences: implications for comparative plant demography. The American Naturalist 176: 710-722
- Tuljapurkar. 1990. Population Dynamics in Variable Environments. Springer. New York, USA.
- Zuidema, Jongejans, Chien, During & Schieving. 2010. Integral Projection Models for trees: a new parameterization and a validation of model output. Journal of Ecology 98: 345-355.