

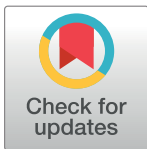
RESEARCH ARTICLE

AMModels: An R package for storing models, data, and metadata to facilitate adaptive management

Therese M. Donovan^{1*}, Jonathan E. Katz²

1 U.S. Geological Survey, Vermont Cooperative Fish and Wildlife Research Unit, Rubenstein School of Environment and Natural Resources, University of Vermont, Burlington, Vermont, United States of America, **2** Vermont Cooperative Fish and Wildlife Research Unit, Rubenstein School of Environment and Natural Resources, University of Vermont, Burlington, Vermont, United States of America

* tdonovan@uvm.edu



Abstract

Agencies are increasingly called upon to implement their natural resource management programs within an adaptive management (AM) framework. This article provides the background and motivation for the R package, **AMModels**. **AMModels** was developed under R version 3.2.2. The overall goal of **AMModels** is simple: To codify knowledge in the form of models and to store it, along with models generated from numerous analyses and datasets that may come our way, so that it can be used or recalled in the future. **AMModels** facilitates this process by storing all models and datasets in a single object that can be saved to an .RData file and routinely augmented to track changes in knowledge through time. Through this process, **AMModels** allows the capture, development, sharing, and use of knowledge that may help organizations achieve their mission. While **AMModels** was designed to facilitate adaptive management, its utility is far more general. Many R packages exist for creating and summarizing models, but to our knowledge, **AMModels** is the only package dedicated not to the mechanics of analysis but to organizing analysis inputs, analysis outputs, and preserving descriptive metadata. We anticipate that this package will assist users hoping to preserve the key elements of an analysis so they may be more confidently revisited at a later date.

OPEN ACCESS

Citation: Donovan TM, Katz JE (2018) AMModels: An R package for storing models, data, and metadata to facilitate adaptive management. PLoS ONE 13(2): e0188966. <https://doi.org/10.1371/journal.pone.0188966>

Editor: Esmaeil Ebrahimie, Flinders University, AUSTRALIA

Received: June 6, 2017

Accepted: November 16, 2017

Published: February 28, 2018

Copyright: This is an open access article, free of all copyright, and may be freely reproduced, distributed, transmitted, modified, built upon, or otherwise used by anyone for any lawful purpose. The work is made available under the [Creative Commons CC0](https://creativecommons.org/licenses/by/4.0/) public domain dedication.

Data Availability Statement: All relevant data are within the paper and its Supporting Information files.

Funding: This work was supported by the U. S. Bureau of Land Management.

Competing interests: The authors have declared that no competing interests exist.

Introduction and motivation

The R package, **AMModels** [1], is a tool for storing models, data, and metadata to facilitate adaptive management. Agencies are increasingly called upon to implement their natural resource management programs within an adaptive management (AM) framework [2–5]. Adaptive management is a key initiative for the U.S. Department of Interior, which offers the following definition [6]:

"Adaptive management promotes flexible decision making that can be adjusted in the face of uncertainties as outcomes from management actions and other events become better understood. Careful monitoring of these outcomes both advances scientific understanding and helps adjust policies or operations as part of an iterative learning process. Adaptive management

also recognizes the importance of natural variability in contributing to ecological resilience and productivity. It is not a ‘trial and error’ process, but rather emphasizes learning while doing. Adaptive management does not represent an end in itself, but rather a means to more effective decisions and enhanced benefits. Its true measure is in how well it helps meet environmental, social, and economic goals, increases scientific knowledge, and reduces tensions among stakeholders."

Williams and Brown [6] explain that this definition emphasizes three key AM principles:

1. Uncertainty about management impacts
2. Iterative learning to reduce uncertainty
3. Improved management as a result of learning.

Models are a crucial part of this process. Natural resource managers are concerned with understanding the dynamics of a system of interest. The system state at time t is influenced by external drivers and management actions, which yield the state of the system at time $t + 1$ (Fig 1).

At the heart of this diagram are **models**, which represent our current understanding of how a system of interest works. If a system is perfectly understood, a model can perfectly predict the state of the system at time $t + 1$, given the state at time t and the environmental and management forces that act on the system. However, environmental systems are rarely, if ever, understood with certainty. Sources of uncertainty that are highlighted in Fig 1 include:

1. Environmental Variation (also called aleatory uncertainty)—environmental factors that have inherent natural variability, such as rainfall and temperature, that affect the system state. Environmental variation can be better understood by additional study, but additional study cannot reduce it.
2. Partial Controllability—the uncertainty about how management actions will affect the system.
3. Structural Uncertainty—uncertainty with the model itself. This may arise when the model does not adequately reflect the system dynamics and/or when the parameters within a model are not well understood.

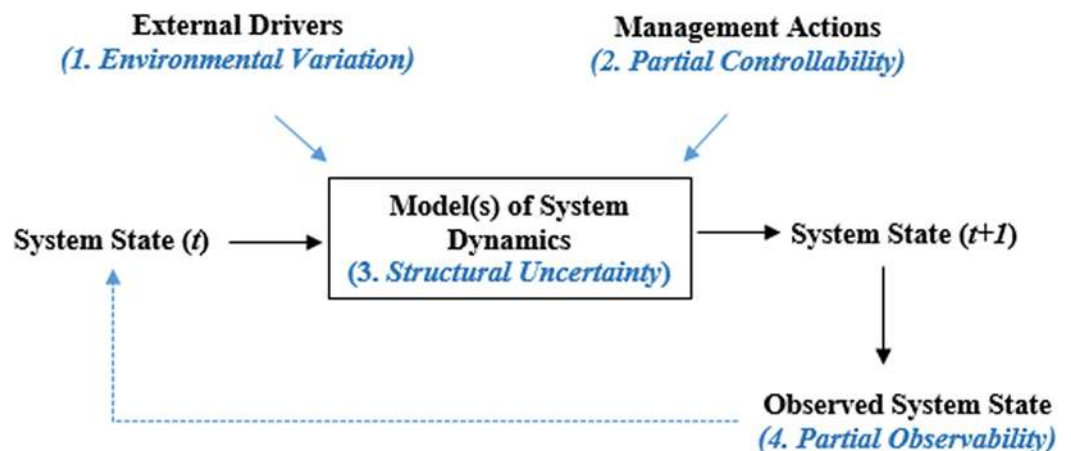


Fig 1. Components of an adaptive management system. Adapted from: National Conservation Training Center ALC3176; Nichols, Runge, and Johnson.

<https://doi.org/10.1371/journal.pone.0188966.g001>

4. Partial Observability—uncertainty that arises when humans cannot observe the system state perfectly. This, in turn, may contribute to structural uncertainty.

The latter three sources of variation can be reduced by scientific study, and reducing uncertainty may lead to better decision making.

In an idealized adaptive management framework, the state of the system is sufficiently monitored to detect changes of interest. When that state appears to be out of sync with an agency's objectives, the agency considers a suite of possible management actions that can move the system towards its intended target. The model, though imperfect, is then used to help predict how each alternative may affect the state, pointing to the action(s) that best achieve agency goals. The selected management action(s), together with environmental drivers, then move the system to system state $t + 1$, and the process reiterates. A central premise of adaptive management is that the model of system dynamics can be continually improved with each passing iteration. Thus, the model of system dynamics changes as our understanding of the system improves [3].

Although the term adaptive management is liberally used in resource management, a large gap between the theory and the practice of adaptive management can exist [7–13]. Not all natural resource management problems fall within the adaptive management umbrella, and elucidating shared goals and objectives among stakeholders can be extremely challenging [14,15]. Even if objectives are well defined, in practice it can be difficult to understand how to implement an adaptive management program that seamlessly integrates data collection, models, and analysis. To aid at least part of this process, we developed the R package, **AMModels**, as a vehicle for organizing data, models, and metadata using the open source modeling platform, R [16]. R can be downloaded from the Comprehensive R Archive Network website (<https://cran.r-project.org/>).

Objectives

This paper outlines the R package **AMModels**, which is foundational to our other AM packages [17,18]. The overall goal of **AMModels** is simple: To codify knowledge in the form of models and to store it, along with models generated from numerous analyses and datasets that may come our way, so that it can be used for future purposes. **AMModels** facilitates this process by storing all models and datasets in a single object that can be saved to an .RData file and routinely augmented to track changes in knowledge through time.

We begin by orienting the reader to R models and model outputs, which can be highly variable from package to package and function to function. We illustrate 3 modeling approaches. First, we analyze plant weight data with the function `lm`, which produces a linear model whose output is stored in an S3 object of class **lm**. Second, we analyze occupancy data of chorus frogs, which produces an occupancy model whose output is stored in an S4 object of class **unmarkedFitOccu**. Finally, we analyze fictitious apple data using a Bayesian analysis, in which the model inputs and outputs are not stored in an object with an analysis-specific class. In these examples, we do not dwell on the methods involved but rather focus on the structure of the analytical inputs and output. With this background, we then introduce the core functionality within the package, **AMModels**, including a Shiny application called **AM Model Manager** which enables the use of these functions in user-friendly format.

Models in R

Models come in a rich variety of flavors, but a common flavor is a model that is generated from a statistical analysis.

Example 1: Linear models

As an example of a model generated from previous analysis, consider the code from the `lm` helpfile featuring plant weights as a function of a treatment. We use this example generically; in the context of adaptive management, a resource manager may be interested in how a treatment or management action affects a resource of interest (e.g., plant weight). Here, a linear model is fit to the observed data, where weight is predicted by the treatment effect (group).

```
require(graphics)

# Annette Dobson (1990) 'An Introduction to Generalized Linear
Models'.

# Page 9: Plant Weight Data.

ctl <- c(4.17, 5.58, 5.18, 6.11, 4.5, 4.61, 5.17, 4.53, 5.33, 5.14)
trt <- c(4.81, 4.17, 4.41, 3.59, 5.87, 3.83, 6.03, 4.89, 4.32, 4.69)
group <- gl(n = 2, k = 10, length = 20, labels = c("Ctl", "Trt"))
weight <- c(ctl, trt)

# store the data within a dataframe
data <- data.frame(weight, group)

# run the analysis
lm.D9 <- lm(weight ~ group, data = data)

# look at the class of lm.D9
class(lm.D9)

[1] "lm"
```

Model results are stored in an object of class `lm`, which is structured as a list with 13 elements. This can be verified with the `str` function. A quick way to see hints about the contents of this object is to use the `names` function.

```
# look at the names of the lm components
names(lm.D9)

[1] "coefficients" "residuals" "effects" "rank"
[5] "fitted.values" "assign" "qr" "df.residual"
[9] "contrasts" "xlevels" "call" "terms"
[13] "model"
```

Typically, analytical outputs contain not only the model results, but additional information regarding the analysis. For example, the data that were input to the analysis are stored in the list item named **model**.

```
# extract the analysis inputs (data)
lm.D9["model"]
$model
```

	weight	group
1	4.17	Ctl
2	5.58	Ctl
3	5.18	Ctl
4	6.11	Ctl
5	4.50	Ctl
6	4.61	Ctl
7	5.17	Ctl
8	4.53	Ctl
9	5.33	Ctl
10	5.14	Ctl
11	4.81	Trt
12	4.17	Trt
13	4.41	Trt
14	3.59	Trt
15	5.87	Trt
16	3.83	Trt
17	6.03	Trt
18	4.89	Trt
19	4.32	Trt
20	4.69	Trt

It is generally unnecessary to use such list indexing to work with the `lm` output, as the authors of the `lm` function provide numerous functions (methods) for extracting and manipulating content stored in an object of class **lm**. The names of the methods that work on a particular class can be displayed with the `methods` function:

```
# return the functions that operate on an object with class lm
methods(class = "lm")

[1]  add1      alias      anova      case.names
[5]  coerce    confint    cooks.distance  deviance
[9]  dfbeta    dfbetas    drop1      dummy.coef
[13] effects    extractAIC  family     formula
[17] hatvalues  influence   initialize  kappa
[21] labels     logLik     model.frame  model.matrix
[25] nobs       plot       predict     print
[29] proj       qr         residuals   rstandard
[33] rstudent   show       simulate    slotsFromS3
[37] summary    variable.names  vcov

see '?methods' for accessing help and source code
```

For instance, the `residuals` method will extract the "residuals," the `predict` method will extract the "fitted.values" from an `lm` object, the `model.frame` method will return the data input, and the `summary` method will print the model residuals, coefficients and their associated statistics.

```
summary(lm.D9)

Call:
lm(formula = weight ~ group, data = data)

Residuals:
      Min       1Q   Median       3Q      Max
-1.0710 -0.4938  0.0685  0.2462  1.3690

Coefficients:
      Estimate Std. Error  tvalue Pr(>|t|)
(Intercept)  5.0320      0.2202  22.850  9.55e-15 ***
groupTrt     -0.3710      0.3114  -1.191  0.249

Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.6964 on 18 degrees of freedom
Multiple R-squared:  0.07308,    Adjusted R-squared:
0.02158
F-statistic: 1.419 on 1 and 18 DF,    p-value: 0.249
```

The output of the `lm` function can be expressed as a basic linear model with the form:

$$Y_i = \beta_0 + \beta_1 * X_i + \epsilon_i.$$

We can substitute in our variable names and coefficients:

$$\text{weight}_i = 5.0320 - 0.3710 * \text{treatment}_i.$$

If a resource manager is interested in a system state (plant weight at time t), this model can be used to help determine if a management action (the treatment) should be used to shift the system to a new state at time $t + 1$. Average plant weight for the control group was 5.0320 units, while average plant weight for the treatment group was $5.0320 - 0.3710 = 4.661$. These averages are not far apart, and the model results indicate that the treatment may not yield a strong change in plant weight.

An important part of the `lm` output is the model ‘fit,’ which conveys how well the linear model fits the observed data. In this case, the model does not fit very well (adjusted R-squared = 0.02158). Additionally, the standard errors associated with each coefficient reflect our uncertainty in the parameter estimates, and thus influence our belief that the treatment does not strongly affect plant weight. Both represent a form of structural uncertainty in the model. And because treatment, a management action, is included within this model, partial controllability is another source of uncertainty.

To summarize this example, the analytical inputs are stored as a dataframe, and the analytical outputs are returned as an object of class `lm`.

Example 2. Occupancy modeling

Model uncertainty may also result from the structural form of the model itself. To illustrate, we now consider two alternative models from a single season occupancy analysis for a wildlife species of interest [19]. The primary input for a single season occupancy analysis consists of raw survey data across multiple study sites in the form of detection and non-detection data, where 1 means the species was detected on a given survey at a given site and 0 means the species was not detected. Additional inputs include covariates associated with each site (such as habitat type or elevation) and covariates associated with the conditions associated with each survey (such as date, time, weather conditions, or observer). The analysis returns maximum likelihood estimates for occupancy (the probability that a site is occupied) and detection (the probability that a species will be detected, given it is present on a site).

The single season occupancy analysis can be run with the `occu` function in the R package, `unmarked` [20]. Here, we use the code in the `occu` helpfile to run a single-season occupancy analysis on chorus frog (*Pseudacris feriarum*) occupancy at 130 sites in Delaware [21].

```
# load package unmarked
suppressPackageStartupMessages(library(unmarked))
# load the chorus frog data
data(frogs)
# create an unmarked frame
pferUMF <- unmarkedFrameOccu(pfer.bin)
# get the number of sites in which frogs were surveyed
```

```

num.sites <- numSites(pferUMF)
# add a fake site level covariate called sitevar1 for illustration
# specify a seed for a random number generator; used for
# reproducibility.
set.seed(20)
site.sums <- rowSums(pferUMF@y, na.rm = T)
sitevar1 = round(rnorm(n = num.sites, mean = ifelse(test = site.
sums > 0, yes = 15, no = 10), sd = 3), 2)
siteCovs(pferUMF) <- data.frame(sitevar1)
# add a fake observation covariate called obsvar1 for illustration
obsCovs(pferUMF) <- data.frame(obsvar1 = c(rep(1:3, num.sites)))
# look at the first 10 records of the resulting unmarkedFrame
object.
head(pferUMF, n = 10)

```

Data frame representation of unmarkedFrame object.

	y.1	y.2	y.3	sitevar1	obsvar1.1	obsvar1.2	obsvar1.3
1	1	0	NA	18.49	1	2	3
2	1	0	0	13.24	1	2	3
3	1	0	0	20.36	1	2	3
4	1	0	0	11.00	1	2	3
5	0	0	0	8.66	1	2	3
6	0	0	0	11.71	1	2	3
7	0	0	0	1.33	1	2	3
8	0	0	0	7.39	1	2	3
9	1	0	0	13.61	1	2	3
10	0	0	0	8.33	1	2	3

The **unmarkedFrame** object is the input needed to run a single season occupancy analysis with the `occu` function. In the (human-readable) data frame representation displayed above, the first three columns (labeled `y.1`, `y.2`, and `y.3`) give the results of the chorus frog surveys, of which there are three. For instance, on site 1, chorus frogs were detected on survey 1, not detected on survey 2, and survey 3 was not conducted, yielding 1, 0, NA in row one, columns one through three of the dataset. On site 2, chorus frogs were detected on survey 1, not detected on survey 2, and not detected on survey 3, yielding 1, 0, 0 in row two.

There is a single site-level covariate in this dataset, named **sitevar1**, which was simulated such that sites with detections generally had higher **sitevar1** values compared to sites with no detections. For example, site 1's **sitevar1** value is 18.49, while site 6's **sitevar1** value is 11.71.

There is a single detection covariate in this dataset, named **obsvar1**. Notice that there are three columns associated with **obsvar1**, representing the value of **obsvar1** for each of the three surveys. Here, **obsvar1**'s value is the survey number (1, 2, or 3), which presumably ranks the survey date.

The `occu` function can be used to analyze the **unmarkedFrame** object with a single-season occupancy model. This function requires that two models are specified as a double right-hand side formula: one component for detection (which is specified first) and one for occupancy. Here, we run the occupancy model when detection is a function of **obsvar1** and occupancy is a function of **sitevar1**, and point to the **unmarkedFrame** called **pferUMF** as the required data input.

```
# fit a single-season occupancy model where detection is a function
of obsvar1 and occupancy is a function of sitevar1
fm1 <- occu(formula = ~obsvar1 ~ sitevar1, data = pferUMF)
```

The output of this analysis, named here as **fm1**, is an object is of class **unmarkedFitOccu**, which we can verify with the `class` function. The "package" attribute identifies the R package that produced this object, which will be handy to jog our human memory when we store models for future re-use.

```
# get the class of fm1
class(fm1)
[1] "unmarkedFitOccu"
attr(,"package")
[1] "unmarked"
```

The output contains the raw information needed to draw conclusions about chorus frog occupancy rates and the covariates that (may) affect occupancy state and detection. The output is stored in 13 slots, and includes not only the model output, but also the details about the analysis, including the two primary inputs (formula and data).

```
# get the slots names associated with fm1 output
slotNames(fm1)
[1] "knownOcc" "fitType" "call"
[4] "formula" "data" "sitesRemoved"
[7] "estimates" "AIC" "opt"
[10] "negLogLike" "nllFun" "bootstrapSamples"
[13] "covMatBS"
```

As with the `lm` output, the authors of **unmarked** provide numerous methods for summarizing and extracting results from an **unmarkedFitOccu** object. For example, the `summary` method will return the model coefficients, the AIC, and other information for evaluating the model.

```
# run unmarked's summary function
summary(fm1)
Call:
occu(formula = ~obsvar1 ~ sitevar1, data = pferUMF)
Occupancy (logit-scale):
              Estimate SE      z    P(>|z|)
(Intercept) -7.340   1.480 -4.96 7.08e-07
sitevar1     0.575   0.124  4.63 3.71e-06

Detection (logit-scale):
              Estimate SE      z    P(>|z|)
(Intercept)  4.71    1.083  4.35 1.38e-05
obsvar1     -3.57    0.665 -5.37 8.06e-08

AIC: 159.3813
Number of sites: 130
optim convergence code: 0
optim iterations: 38
Bootstrap iterations: 0
```

The `occu` function models on the log-odds (logit) scale, which is required because occupancy and detection rates are probabilities that are constrained between 0 and 1. Thus, the coefficients of an analysis output are on the logit scale. The chorus frog occupancy model has the form:

$$\text{logit}(\text{occupancy})_i = -7.340 + 0.575 * \text{sitevar1}_i$$

This model (if it is shown to be adequate) can be of great utility to natural resource managers interested in managing chorus frogs. The model of chorus frog system occupancy suggests that sites with **sitevar1** levels $> \sim 15$ units have a high chance of being occupied. If **sitevar1** can be manipulated by management activities, this model may allow resource managers to predict how the state of the system (occupancy) at time t can be shifted to a desired state at time $t + 1$ (Fig 1) by manipulating **sitevar1** across sites.

We can back-transform the logit score to a probability with the log-odds transformation, and use the code on the following page to plot our results (Fig 2).

Chorus Frog Occupancy as a Function of Sitevar1

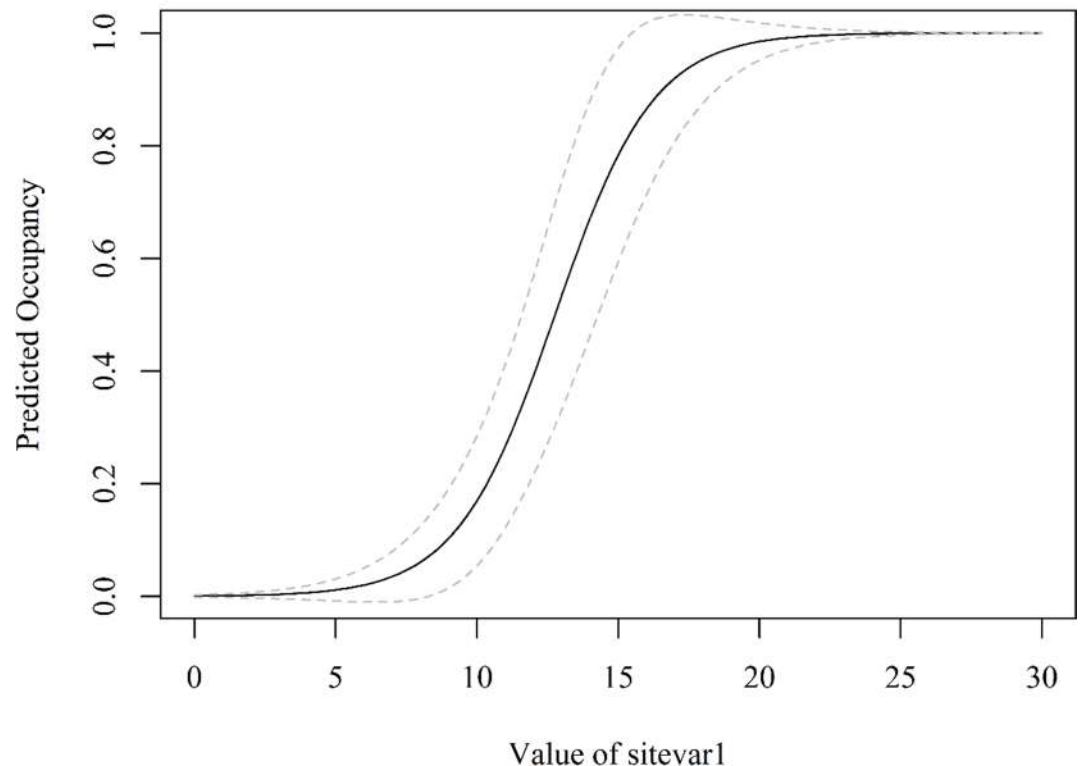


Fig 2. Chorus frog occupancy as a function of the simulated covariate, sitevar1.

<https://doi.org/10.1371/journal.pone.0188966.g002>

```
# create a matrix with a range of sitecov values
data <- cbind(1, seq(from = 0, to = 30, length.out = 100))

# calculate the linear score (logit) across a range of sitecov
values
lc <- linearComb(obj = fm1["state"], data)

# use unmarked's backTransform function to convert the linear score
(logit)
# to probability
btlc <- backTransform(lc)

# convert to dataframe and add upper and lower confidence intervals
btlc <- as(btlc, "data.frame")
btlc$uci <- btlc$Estimate + 1.96 * btlc$SE
btlc$lci <- btlc$Estimate - 1.96 * btlc$SE

# plot the predicted occupancy rates
par(family = "serif")
```

```
plot(x = btlc$sitevar1, y = btlc$Estimate, type = "l", main = "Chorus Frog Occupancy as a Function of Sitevar1",
      xlab = "Value of sitevar1", ylab = "Predicted Occupancy")
# add uncertainty in the form of confidence interval lines
lines(x = btlc$sitevar1, y = btlc$lci, lty = "dashed", col = "gray")
lines(x = btlc$sitevar1, y = btlc$uci, lty = "dashed", col = "gray")
```

The `occu` outputs also contain a detection model, which, based on the model output, now has the form:

$$\text{logit}(\text{detection})_i = 4.71 + -3.57 * \text{obsvar1}_i$$

This model (if it is shown to be adequate) suggests that as `obsvar1` increases, the detection probability of chorus frogs decreases (a scan of the raw data shows that most frogs were detected in survey 1 of the study). This model attempts to deal with partial observability in [Fig 1](#). That is, the analyst attempts to discover those covariates that affect observability so that this source of uncertainty can be reduced. The `occu` model results show that conditions associated with first chorus frog surveys resulted in much higher detection probability than in later surveys. Although this result will probably not help identify management actions that push the system state (occupancy), this information can be used to improve the accuracy of the monitoring program, and hence reduce partial observability ([Fig 1](#)). This, in turn, may lead to a better model of system dynamics.

In most cases, multiple models are evaluated and compared to determine which model(s) best explain the observed, raw data [\[22\]](#). To wit, we now run a second occupancy analysis on the chorus frog data, in which both detection and occupancy have no covariates (i.e., only the intercepts will be estimated):

```
# fit a single season occupancy model with no covariates
fm2 <- occu(formula = ~1 ~ 1, data = pferUMF)
```

We can now compare the two models with the `fitList` and `modSel` functions in **unmarked**.

```
# create a list of models that will be compared
fl <- fitList(fm1. = fm1, fm2. = fm2)
# perform model selection analysis
ms <- modSel(object = fl, nullmod = "fm2.")
# view the results of the analysis
```

```
ms
```

	nPars	AIC	delta	AICwt	cumltvWt	Rsq
fm1.	4	159.38	0.00	1.0e+00	1.00	0.65
fm2.	2	261.33	101.95	7.3e-23	1.00	0.00

Without getting into the weeds [22], the AIC weights (AICwt) from this analysis suggest that model 1 (**fm1**) has 100% support, meaning that the model with no covariates (**fm2**) has 0 support. In other words, the structure of **fm2** has so much structural uncertainty compared to **fm1** that it serves no purpose in chorus frog management. We would still like some evidence that **fm1** is a good ‘fit’ to our observed field data, which we can do with the MacKenzie and Bailey Goodness-of-fit Test [19]. In R, this test is implemented by the `mb.gof.test` function in the package **AICcmodavg** [23], typically executed with 1000’s of trials.

```
# load the package AICcmodavg
library(AICcmodavg)
# run the goodness of fit test
AICcmodavg::mb.gof.test(mod = fm1, nsim = 100, plot.hist = FALSE)
MacKenzie and Bailey goodness-of-fit for single-season occupancy
model
Pearson chi-square table:
      Cohort  Observed  Expected  Chi-square
000      0         58      66.22      1.02
010      0          4       0.77     13.43
100      0         35     27.49      2.05
00NA     1         12      9.19      0.86
10NA     1          1      3.41      1.70
ONANA    2         17     14.58      0.40
1NANA    2          3      5.42      1.08

Chi-square statistic = 23.4566
Number of bootstrap samples = 100
P-value = 0.04
Quantiles of bootstrapped statistics:
      0%   25%  50%  75% 100%
0.22  2.13  3.26  6.00 74.71

Estimate of c-hat = 3.99
```

Again, without dwelling on details, this analysis suggests that even **fm1** does not adequately explain the observed data (with a very high overall Chi-square statistic of ~ 23 , meaning our observed and expected results are not very well matched). Keep in mind that we simulated the covariates in this model, so a lack of fit is not surprising. However, if the data were not simulated, this goodness of fit test suggests that **fm1** has significant structural uncertainty, and we should look for a better occupancy model, or even a different modeling approach.

To summarize this example, the analysis inputs are stored in an object of class **unmarked-Frame**, and the analysis outputs are returned as an object of class **unmarkedFitOccu**.

Example 3. Bayesian modeling

Our final example is a Bayesian approach, which is an ideal modeling framework for adaptive management because it codifies learning through time. The International Society of Bayesian Analysis describe the Bayesian paradigm, which can be summarized as follows. First, current knowledge about the model parameters of interest is expressed by placing a ‘prior probability distribution’ on the parameters, which is often written as

$$p(\theta).$$

Second, when new data y become available, the likelihood of observing the data given the model parameters is calculated, which is often expressed as

$$p(y|\theta).$$

Finally, the likelihood is then combined with the prior to produce an updated probability distribution called the ‘posterior distribution,’ upon which all Bayesian inference is based. The posterior distribution is often written as:

$$p(\theta|y).$$

Bayes’ Theorem is an elementary identity in probability theory, and states how the prior distribution is updated to the posterior distribution in light of new information, y . More precisely,

$$p(\theta|y) = \frac{p(\theta)p(y|\theta)}{\int p(\theta)p(y|\theta)d\theta}.$$

Normally a Bayesian analysis in R would be conducted via a package such as **rjags** [24], in which the analytical inputs are stored in an object of class **jags** and the model output is stored in an object of class **mcmc.list**. Here, though, to illustrate the complete flexibility of **AMModels**, we will store inputs and outputs as typical S3 objects.

Suppose we have a need to estimate the probability of that apples in an orchard will be infested by the apple maggot. Let’s call this probability p . With a Bayesian analysis, the analyst must provide **prior** distributions for each parameter of interest, and we are focused on the single parameter, p . We can represent our beliefs in alternative values of p with a beta distribution—a probability density function that is controlled by two parameters, normally called alpha (α) and beta (β). Let’s first graph our prior distribution for the unknown parameter, p (Fig 3):

```
# establish prior distribution for the unknown parameter, p (a beta
# distribution)
alpha.prior <- 2
```

```

beta.prior <- 3
p <- seq(from = 0, to = 1, by = 0.01)
prior.density <- dbeta(p, shape1 = alpha.prior, shape2 = beta.prior, ncp = 0,
log = FALSE)
# graph the prior distribution
par(family = "serif")
plot(x = p, y = prior.density, type = "l", ylim = c(0, 10), ylab =
"Weight (Density)", xlab = "Hypotheses for Infestation Probability,
p", col = "red", lwd = 3, main = "Prior Distribution")

```

Prior Distribution

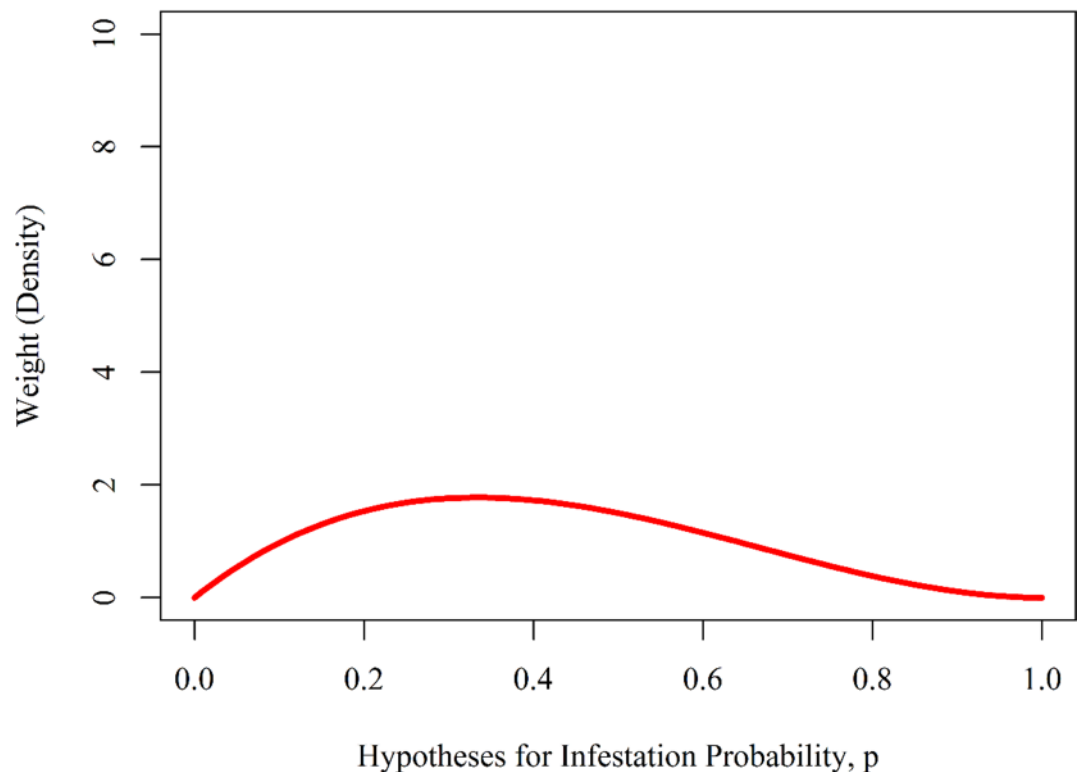


Fig 3. The prior distribution for infestation rate, p , is a beta distribution with $\alpha = 2$ and $\beta = 3$.

<https://doi.org/10.1371/journal.pone.0188966.g003>

Our model for p is fully represented by a beta distribution with parameters $\alpha = 2$ and $\beta = 3$. This particular model suggests that there is quite a bit of uncertainty regarding p .

In an adaptive management framework, the goal is to reduce this uncertainty. For example, if we collect new data on infestation rates, we can use Bayes' Theorem and update our beliefs in alternative hypotheses for p . Suppose we collect a sample of 100 apples from our orchard, and that 25 of them are infested. We can now combine this new data, along with the

parameters of the prior distribution for p , to generate a posterior distribution that reflects our updated knowledge regarding infestation probability. For this particular problem, the parameters for this posterior distribution can be calculated quickly with the following equations, where n is the total number of apples in our sample, and y is the total number of infected apples:

- $\alpha_{\text{posterior}} = \alpha_{\text{prior}} + y = 2 + 25 = 27$
- $\beta_{\text{posterior}} = \beta_{\text{prior}} + n - y = 3 + 100 - 25 = 78$

A graph of the prior distribution and the posterior distribution can be displayed in [Fig 4](#) using the code below: ([Fig 4](#)).

```
# establish posterior distribution for the unknown parameter, p (a
beta
# distribution)
alpha.posterior <- 27
beta.posterior <- 78
# calculate the densities associated with the updated distribution
posterior.density <- dbeta(p, shape1 = alpha.posterior,
shape2 = beta.posterior, ncp = 0, log = FALSE)
# plot the prior distribution
par(family = "serif")
plot(x = p, y = prior.density, xlim = c(0, 1), ylim = c(0, 10), type
= "l",
ylab = "Weight (Density)", xlab = "Hypotheses for Infestation Proba-
bility, p", main = "Prior and Posterior Distributions", col = "red",
lwd = 3)
# add the posterior distribution
lines(x = p, y = posterior.density, col = "blue", lwd = 3)
# add legend
legend("topright", legend = c(expression(paste("Prior Distribution:
", alpha, " = 2, ", beta, " = 3")), expression(paste("Posterior Dis-
tribution: ",
alpha, " = 27, ", beta, " = 78"))), lty = 1, col = c("red", "blue"),
bty = "n", cex = 0.75)
```

Notice how the new data has yielded updated knowledge about infestation probability, considerably reducing our uncertainty. If you were to continue collecting data, you would now use the parameters from the posterior distribution as the prior distribution, combine it with another set of new data, and then generate a third, updated model for apple infestation. These are valuable models that reflect the current knowledge of apple infestation and through time, even though they do not link to management activities or to environmental drivers.

Unlike our previous two examples, the inputs and outputs for this analysis are not shaped or stored in dedicated objects such as `unmarkedFitOccu`. Here, to make the simple point that

Prior and Posterior Distributions

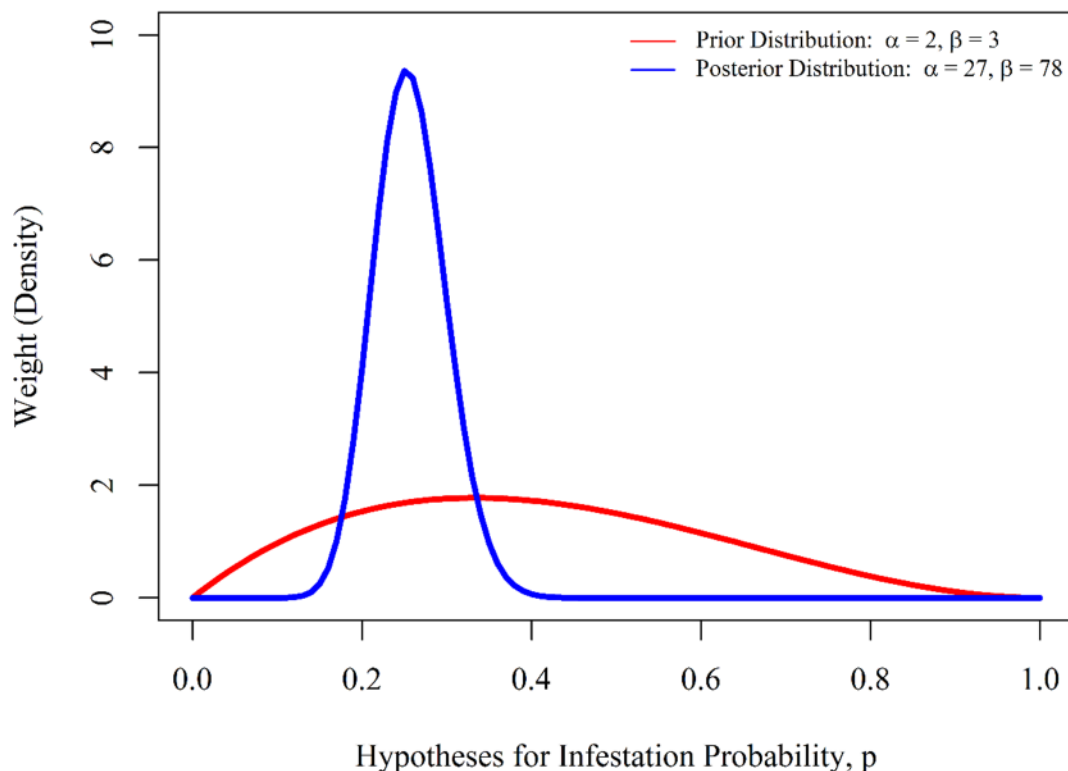


Fig 4. The prior and posterior distributions for infestation probability, p.

<https://doi.org/10.1371/journal.pone.0188966.g004>

AMModels can store user-defined analysis inputs and outputs, we will store our apple analysis as a list, and link the second analysis with the first by specifying a list element named "prior" (note that, in this hypothetical example, **apple.m2** is connected with **apple.m1** via a list element named "prior").

```
# create model1 as a list; coefficients
apple.m1 <- list(name = "apple.m1", distribution = "beta", prior =
list(alpha = 2, beta = 3))

# create model2 as a list; coefficients
apple.m2 <- list(name = "apple.m2", distribution = "beta", prior =
"apple.m1", data = c(y = 25, n = 100), posterior = list(alpha = 27,
beta = 78))

# look at the structure apple.m1
str(apple.m1)

List of 3
 $ name      : chr "apple.m1"
 $ distribution: chr "beta"
```

```
$ prior      : List of 2
  ..$ alpha: num 2
  ..$ beta:  num 3
# look at the structure apple.m2
str(apple.m2)
List of 5
 $ name      : chr "apple.m2"
 $ distribution: chr "beta"
 $ prior     : chr "apple.m1"
 $ data      : Named num [1:2] 25 100
  ..- attr(*, "names") = chr [1:2] "y" "n"
 $ posterior  : List of 2
  ..$ alpha: num 27
  ..$ beta  : num 78
```

Models in Adaptive Management

An enormous variety of analytical approaches exist to meet a variety of needs (e.g., [25–31]). The three examples are intended to simply highlight a few key points with respect to analytical inputs and outputs in R in relation to adaptive management:

1. Analytical inputs to R functions are unique to the analytical framework (e.g. Example 1's input was a dataframe while Example 2's input was an **unmarkedFrame**).
2. Analytical outputs from R functions are also unique to the analytical framework (e.g., Example 1's output was class **lm** while Example 2's output was class **unmarkedFitOccu**).
3. Outputs that are models represent a state of knowledge.
4. All models include elements of uncertainty. In the first two examples provided, there is uncertainty in the coefficient estimates, and there is uncertainty in the model structure.
5. Models may be a primary output from an analysis, but they may also be an input for future analyses (e.g., Example 3).
6. Information associated with an analysis itself (i.e., the analysis metadata) can preserve the context of the analysis so that the analysis may be more confidently revisited at a later date.

In an adaptive management program, the models must be explicit, and models may be modified through time as more and more is learned about the system being managed. The notion of adaptive management is to acknowledge uncertainty at every turn, and reduce this uncertainty by careful monitoring and study over time.

The R package, AMModels

The overall goal of **AMModels** is straight-forward: To codify knowledge in the form of models and to store it, along with models generated from numerous analyses and datasets that may come our way, so that it can be used in the future. **AMModels** facilitates this process by storing analysis inputs, outputs, and metadata in a single object that can be repeatedly updated to track changes in knowledge through time.

In this section, we demonstrate the core functionality and typical workflow of **AMModels**. Many of these examples are highlighted in the package's helpfiles. In the final section of this paper, we introduce the **AM Model Manager**, a Shiny application [32] that simplifies the storage and retrieval of models and associated datasets.

```
# install.packages("AMModels")  
  
# load AMModels  
  
library(AMModels)
```

amModelLib Object

A central idea of **AMModels** is that all models and datasets, along with their metadata, are stored in an **amModelLib** object, where "Lib" stands for "library." This is an S4 object that can be updated and used repeatedly as a way of carting around knowledge in the form of models. Think of the **amModelLib** as a vault or repository that stores datasets (analysis inputs) and models (analysis outputs). The function `amModelLib` creates the object.

```
# create an amModelLib object  
  
mymodels <- amModelLib(description = "This AM Model Library stores  
models and data.", info = list(owner = "me", email = "me@somewhere.  
com"))  
  
# view the amModelLib  
  
mymodels  
  
Description:  
[1] This AM Model Library stores models and data.  
  
Info:  
  
  owner  
  [1] me  
  
  email  
  [1] me@somewhere.com  
  
  date.created  
  [1] 2017-10-04 07:28:28  
  
Models:
```

```

--- There are no models ---
Data:
--- There are no datasets ---

```

An **amModelLib** objects contain 4 slots, which can be verified with the `slotNames` function.

```

# view the slot names
slotNames(mymodels)
[1] "models"      "data"        "info"        "description"

```

Fig 5 highlights the four slots of the **amModelLib**:

1. **description.** A text field that describes the **amModelLib**. Generally the description is one sentence long. Additional information can be stored in the **info** slot.
2. **info.** A named list that contains additional information about the **amModelLib**, such as the name of the person who creates and maintains the **amModelLib**, and any additional related metadata pertaining to the library itself.
3. **data.** This slot stores datasets, i.e., analysis inputs. Each dataset has a class of **amData**, which consists of two slots: 1) datasets are stored in the data slot, and 2) metadata about the dataset are stored in the metadata slot.

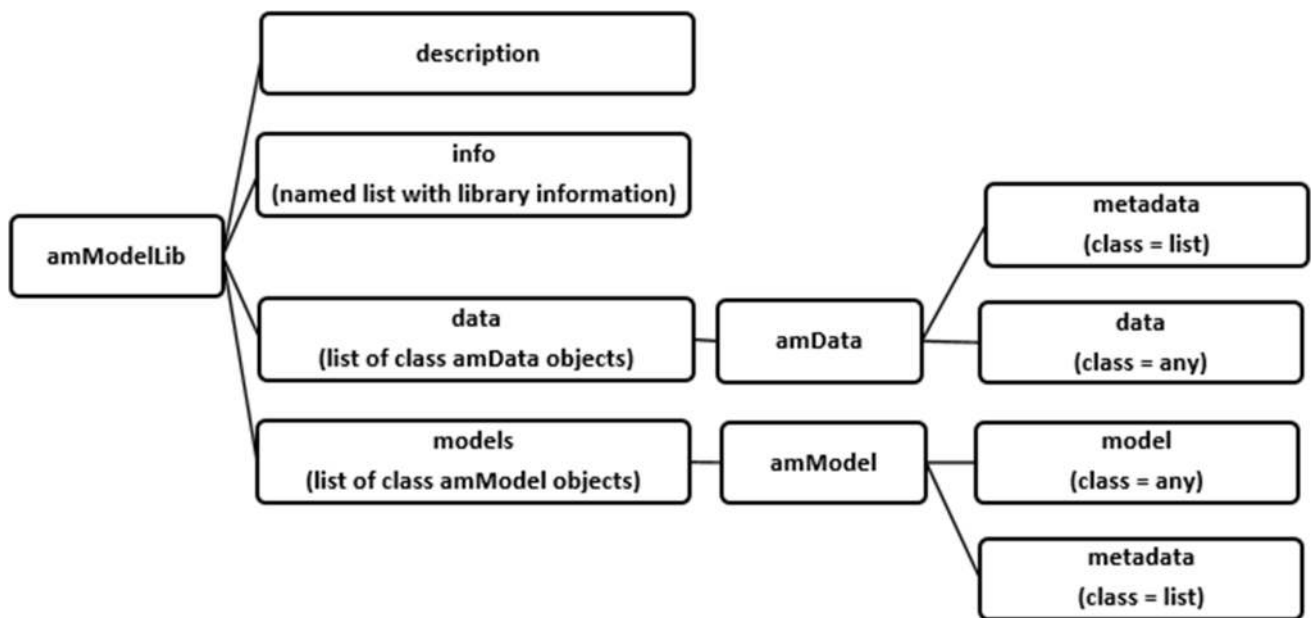


Fig 5. The structure of an **amModelLib object, consisting of four slots.**

<https://doi.org/10.1371/journal.pone.0188966.g005>

- models.** This slot stores models, i.e., analysis outputs. Each model has a class of **amModel**, which consists of two slots: 1) models are stored in the model slot, and 2) metadata about the model are stored in the metadata slot.

Each slot is described below. The package **AMModels** contains functions to create the **amModelLib** object, functions to create objects of class **amModel** and **amData**, and methods to insert, extract, and delete items from the **amModelLib** object. A cheat sheet of the various functions can be found in Supplement 1.

The Description slot. The description slot can be viewed or updated with the function `ammlDesc`.

```
# show the amModelLib description
ammlDesc(amml = mymodels)

[1] "This AM Model Library stores models and data."

# update the description
ammlDesc(amml = mymodels) <- "This AM Model Library stores analysis
inputs (data) and analysis outputs (models) associated with the
AMModels package vignette."

# show the amModelLib description
ammlDesc(amml = mymodels)

[1] "This AM Model Library stores analysis inputs (data) and analy-
sis outputs (models) associated with the AMModels package
vignette."
```

The Info slot. The info slot of an **amModelLib** is a named list that stores information about the model library itself. This list is unlimited in length. Information is stored in name (key) = value syntax, such as:

- owner = "me"
- email = "me@somewhere.com"

The **amModelLib** info can viewed, retrieved, updated, or deleted with the function, `ammlInfo`:

```
# look at the metadata associated with the amModelLib
ammlInfo(amml = mymodels)

$owner

[1] "me"

$email

[1] me@somewhere.com
```

```
$date.created
[1] "2017-10-04 07:28:28"
# extract only the owner
ammlInfo(amml = mymodels, x = "owner")
$owner
[1] "me"
# deletions are done by setting the value associated with a key to
NULL
ammlInfo(amml = mymodels) <- list(date.created = NULL)
# update the owner name to Me, and add a new metadata element called
# organization
ammlInfo(amml = mymodels) <- list(owner = "Me", organization = "My
Organization")
# view the result
ammlInfo(amml = mymodels)
$owner
[1] "Me"
$email
[1] me@somewhere.com
$organization
[1] "My Organization"
```

The Data slot. The data slot of an **amModelLib** object contains a list of datasets (analysis inputs), where each dataset is stored as an object of class **amData**. Although the outputs of many analyses will contain the inputs as well (as with the `lm` and `occu` examples), the purpose of this new class is to allow the user to store metadata associated with the inputs (datasets) to an analysis.

The key functions associated with **amData** objects include:

- `amData`—creates an object of class **amData**.
- `insertAMModelLib`—inserts objects of class **amData** to an **amModelLib** object.
- `rmData`—deletes an **amData** object from an **amModelLib** object.
- `getAMData`—extracts ("checks out") an **amData** object from the **amModelLib** object and returns it to R's global environment.

To illustrate, let's add the plant weight data that we analyzed earlier from the `lm` helpfile. To add the dataset, we first convert the dataset to an **amData** object with the function, `amData`:

```
# Annette Dobson (1990) 'An Introduction to Generalized Linear
Models'.
# Page 9: Plant Weight Data.
ctl <- c(4.17, 5.58, 5.18, 6.11, 4.5, 4.61, 5.17, 4.53, 5.33, 5.14)
trt <- c(4.81, 4.17, 4.41, 3.59, 5.87, 3.83, 6.03, 4.89, 4.32, 4.69)
group <- gl(n = 2, k = 10, length = 20, labels = c("Ctl", "Trt"))
weight <- c(ctl, trt)
# store the data within a dataframe
plant.wt <- data.frame(weight, group)
# create an object of class amData, which includes the dataset and
# corresponding metadata.
plant.data <- amData(data = plant.wt, comment = "Plant dataset from
the lm helpfile.", taxa = "plants")
```

The function `amData` creates an object of class **amData**, which contains two slots. The first slot is called **data**, which stores the data. In this example, the data are stored as a data frame with 20 observations and 2 variables (the actual plant weight dataset).

The second slot is called **metadata**, which contains a list of metadata associated with this dataset. In the example above, we added metadata fields for comment and taxa. Users should create and standardize their own metadata naming conventions with a key-value syntax. The metadata descriptions (values) can be any length. It is up to the user to determine which metadata to store (the values) and their corresponding names. However, the object creation date, "date.created" is automatically added.

Typing in the name of the **amData** object will display its contents.

```
# look at the amData object
plant.data
An object of class "amData"
Slot "data":
      weight      group
1      4.17      Ctl
2      5.58      Ctl
3      5.18      Ctl
4      6.11      Ctl
5      4.50      Ctl
6      4.61      Ctl
```

```

7      5.17      Ctl
8      4.53      Ctl
9      5.33      Ctl
10     5.14      Ctl
11     4.81      Trt
12     4.17      Trt
13     4.41      Trt
14     3.59      Trt
15     5.87      Trt
16     3.83      Trt
17     6.03      Trt
18     4.89      Trt
19     4.32      Trt
20     4.69      Trt

Slot "metadata":
 $comment
 [1] "Plant dataset from the lm helpfile."
 $taxa
 [1] "plants"
 $date
 [1] "2017-10-04 07:28:28"

```

Alternatively, one could see a summary of the object.

```

# look at a summary of an amData object
summary(plant.data)

```

	weight	group
Min.:	3.590	Ctl:10
1stQu.:	4.388	Trt:10
Median:	4.750	
Mean:	4.846	
3rdQu.:	5.218	
Max.:	6.110	

```

--- Metadata ---

```



```

      name      value
1 comment Plant dataset from the lm helpfile.
2 taxa        plants
3 date        2017-10-04 07:28:28

```

Here, the data slot in the **amData** object is occupied by a data frame. Thus, the `summary` function invokes R's `summary` method for data frames and displays the results. The second portion of the `summary` output is an abbreviated look at the metadata.

The function `insertAMModelLib` can be used to insert this object into the **amModelLib**. The **amData** objects to be inserted must be provided as a named list.

```

# insert data to amModelLib as a named list
mymodels <- insertAMModelLib(data = list(plant.data = plant.data),
                             amml = mymodels)

```

Typing in the name of the **amModelLib** returns an abbreviated look at the contents:

```

# show an abbreviated view of the amModelLib
mymodels
Description:
[1] This AM Model Library stores analysis inputs (data) and analysis
      outputs (models) associated with the AMModels package
      vignette.
Info:
  owner
  [1] Me
  email
  [1] me@somewhere.com
  organization
  [1] My Organization
Models:
--- There are no models ---

```

```
Data:
      name      class rows cols package
1 plant.data data.frame    20    2      NA
```

Under the "Data:" section, we see the name of each **amData** object, its class, and the package of origin (if it is readily available). Rows and columns are also provided for objects that contain data frames.

The metadata associated with all **amData** objects or a specific **amData** objects can be viewed or set with the `dataMeta` function.

```
# view all metadata associated with datasets
dataMeta(amml = mymodels)

$plant.data
$plant.data$comment
[1] "Plant dataset from the lm helpfile."
$plant.data$taxa
[1] "plants"
$plant.data$date
[1] "2017-10-04 07:28:28"

# view metadata associated with the amData object, plant.data
dataMeta(amml = mymodels, x = "plant.data")

$comment
[1] "Plant dataset from the lm helpfile."
$taxa
[1] "plants"
$date
[1] "2017-10-04 07:28:28"

# add some additional metadata to an amData object in the form of a
named
# list

dataMeta(amml = mymodels, x = "plant.data") <- list(url = "https://stat.ethz.ch/R-manual/R-devel/library/stats/html/lm.html",
group = "The column named 'group' identifies which group a sample
belongs to, where Ctl = control group and Trt = treatment group.",
weight = "The column named 'weight' provides total biomass of the
sample, in grams.")
```

The `dataMeta` function can also be used to quickly copy existing metadata to a new **amData** object. For instance, agencies or organizations that produce the same analyses year after year can create a new **amData** object that includes an updated dataset, but utilizes the metadata from a previous year's **amData** object.

```
# recall our original plant data is from the lm helpfile
plant.wt <- data.frame(group, weight)
# add a few new records of plant data
new.plant.data <- data.frame(group = c("Ctl", "Trt"), weight = c(
  4.5, 3.83))
# combine data to make an updated plant dataset
updated.plant.data <- rbind(plant.wt, new.plant.data)
# extract the metadata from the existing amData object, plant.data
metadata <- dataMeta(amml = mymodels, x = "plant.data")
# create new amData object that combines the new data with the
existing metadata
updated.plant.data <- amData(data = updated.plant.data, metadata)
# insert the amData object to the AM model library as a named list
mymodels <- insertAMModelLib(data = list(updated.plant.
data = updated.plant.data), amml = mymodels)
# view metadata associated with the new amData object, updated.
plant.data
dataMeta(amml = mymodels, x = "updated.plant.data")
$comment
[1] "Plant dataset from the lm helpfile."
$taxa
[1] "plants"
$date
[1] "2017-10-04 07:28:28"
$url
[1] https://stat.ethz.ch/R-manual/R-devel/library/stats/html/lm.
html
$group
[1] "The column named 'group' identifies which group a sample
belongs to, where Ctl = control group and Trt = treatment group."
$weight
[1] "The column named 'weight' provides total biomass of the sample,
in grams."
```

Multiple **amData** objects can be inserted to an **amModelLib** simultaneously using the same process. For example, below we create a data frame containing our apple infection data (called **apples**), and a second data frame (called **sim.covs**) that is simulated with the function, **simCovariate**.

```
# create apple infestation data frame
apples <- data.frame(n = 100, y = 25)

# create a list of uncorrelated covariates
cov.list <- list(unif1 = list(dist = "runif", min = 0, max = 10,
seed = 334,
round = 0), unif2 = list(dist = "runif", min = 0, max = 10,
seed = 668,
round = 0), norm1 = list(dist = "normal", mean = 10, sd = 2,
seed = 10,
round = 1), norm2 = list(dist = "normal", mean = 50, sd = 10,
seed = 15,
round = 2), beta1 = list(dist = rbeta, shape1 = 2, shape2 = 1,
seed = 1002),
binom1 = list(dist = "bin", size = 20, prob = 0.5, seed = 561),
bern1 = list(dist = "bernoulli", size = 1, prob = 0.5, seed = 6))

# generate a data frame with the covariate list and show
(sim.covs <- simCovariate(cov.list = cov.list, n = 10, add.
yr = TRUE))
```

	unif1	unif2	norm1	norm2	beta1	binom1	bern1	yr
1	7	6	9.1	44.07	0.9008661	8	1	1
2	8	7	9.0	31.52	0.6645892	11	1	2
3	3	4	11.4	39.52	0.9618653	13	1	3
4	1	8	6.5	45.96	0.8872998	8	1	4
5	10	4	9.4	44.17	0.5601055	11	0	5
6	8	7	7.6	43.26	0.9934818	9	1	6
7	6	8	10.9	42.66	0.6639714	11	1	7
8	0	1	11.2	53.89	0.6274681	6	0	8
9	2	3	8.5	51.77	0.6857797	11	0	9
10	3	1	9.5	63.01	0.7370935	11	1	10

To add these two datasets to our **amModelLib**, we first convert each to an object of class **amData** with the function, **amData**, and then insert it to the **amModelLib** with the **insertAMModelLib** function.

```

# create an amData object containing the apple data.
apple.data <- amData(data = apples, comment = "Apple infestation
dataset for Bayesian analysis.")

# create an amData object containing the simulated covariate data.
sim.data <- amData(data = sim.covs, comment = "Simulated covariate
dataset.")

# add the two amData objects to the amModelLib; remember it must be
named list.
mymodels <- insertAMModelLib(data = list(apple.data = apple.data,
sim.data = sim.data), amml = mymodels)

# show the contents; note the library now contains 4 datasets.
mymodels
Description:
[1] This AM Model Library stores analysis inputs (data) and analysis
      outputs (models) associated with the AMModels package
      vignette.
Info:
  owner
  [1] Me
  email
  [1] me@somewhere.com
  organization
  [1] My Organization
Models:
  --- There are no models ---
Data:
  name          class      rows  cols  package
1  plant.data   data.frame  20    2     NA
2  updated.plant.data data.frame  22    2     NA
3  apple.data   data.frame   1    2     NA
4  sim.data     data.frame  10    8     NA

```

Datasets are not limited to data frames. To demonstrate, we now add the chorus frog unmarked frame object to an **amData** object and add it to our **amModelLib**.

```

# load the chorus frog data
data(frogs)

# create an unmarked frame
pferUMF <- unmarkedFrameOccu(pfer.bin)

# get the number of sites in which frogs were surveyed
num.sites <- numSites(pferUMF)

# add a fake site level covariate called sitevar1 for illustration
# specify a seed for a random number generator; used for
reproducibility.
set.seed(20)

site.sums <- rowSums(pferUMF@y, na.rm = T)
sitevar1 = round(rnorm(n = num.sites, mean = ifelse(test = site.
sums > 0, yes = 15, no = 10), sd = 3), 2)

siteCovs(pferUMF) <- data.frame(sitevar1)

# create an object of class amData
frog.data <- amData(data = pferUMF, comment = "Chorus frog dataset
from the package unmarked.", taxa = "Chorus Frog", url = "http://
www.rdocumentation.org/packages/unmarked/versions/0.11-0/topics/
occu")

# the summary function would show the results of the summary method
in package unmarked summary(frog.data)

# add the amData to the amModelLib as a named list
mymodels <- insertAMModelLib(data = list(frog.data = frog.data),
amml = mymodels)

```

Our **amModelLib** now consists of five **amData** objects within the data slot:

```

# show the contents of an amModelLib object
mymodels
Description:
[1] This AM Model Library stores analysis inputs (data) and analysis
      outputs (models) associated with the AMModels package
      vignette.
Info:
  owner
    [1] Me
  email
    [1] me@somewhere.com
  organization

```

```
[1] My Organization
Models:
--- There are no models ---
Data:
  name          class          rows  cols  package
1 plant.data    data.frame      20    2    <NA>
2 updated.plant.data data.frame      22    2    <NA>
3 apple.data    data.frame       1    2    <NA>
4 sim.data      data.frame      10    8    <NA>
5 frog.data     unmarkedFrameOccu NA    NA    unmarked
```

Use the `lsData` function to retrieve the names of the **amData** objects in an **amModelLib**, where **ls** indicates list (in the general sense):

```
# retrieve the names of the amData objects
lsData(mymodels)
[1] "plant.data"          "updated.plant.data" "apple.data"
[4] "sim.data"            "frog.data"
```

For objects of class **amModelLib**, the `summary` function will provide a summary of the contents of the **amModelLib** as shown above, but will also display an abbreviated view of the metadata.

```
# run the summary method an amModelLib object
summary(mymodels)
Description:
[1] This AM Model Library stores analysis inputs (data) and analysis
      outputs (models) associated with the AMModels package
      vignette.
Info:
  owner
```

```

[1] Me
email
[1] me@somewhere.com
organization
[1] My Organization
--- There are no models ---
--- Data Names and Indices ---
  name          class          rows  cols  package
1 plant.data    dataframe         20    2    <NA>
2 updated.plant.data dataframe         22    2    <NA>
3 apple.data    dataframe          1    2    <NA>
4 simdata       dataframe          10    8    <NA>
5 frogdata      unmarkedFrameOccu  NA    NA    unmarked

--- Data metadata ---
[[1]] plant.data
  name      value
1 comment  Plant dataset from the lm helpfile
2 taxa     plants
3 date     2017-10-04 07:28:28
4 url      https://stat.ethz.ch/R-manual/R-devel/library/stats/html/lm ...
5 group    The column named 'group' identifies which group a sample bel ...
6 weight   The column named 'weight' provides total biomass of the samp ...

[[2]] updated.plant.data
  name      value
1 comment  Plant dataset from the lm helpfile
2 taxa     plants
3 date     2017-10-04 07:28:28
4 url      https://stat.ethz.ch/R-manual/R-devel/library/stats/html/lm ...
5 group    The column named 'group' identifies which group a sample bel ...
6 weight   The column named 'weight' provides total biomass of the samp ...

```



```

[[3]] apple.data
  name      value
1  comment  Apple infestation dataset for Bayesian analysis.
2  date     2017-10-04 07:28:28

[[4]] sim.data
  name      value
1  comment  Simulated covariate dataset.
2  date     2017-10-04 07:28:28

[[5]] frog.data
  name      value
1  comment  Chorus frog dataset from the package unmarked.
2  taxa     Chorus Frog
3  url      http://www.rdocumentation.org/packages/unmarked/versions/0.1
4  date     2017-10-04 07:28:29

```

The `summary` output is intended to let the user scan the names and indices of the models and datasets within the `amModelLib` object, and then use the index to quickly find the metadata associated with each object. Notice that the `summary` function truncates the metadata values.

To retrieve a specific `amData` object and return the object to R's global environment in its original class, the `getAMData` accessor function can be used. Let's extract the plant data with the `getAMData` function, which requires the name of the `amData` object. The argument, `as.list` indicates how the object is to be returned: `FALSE` will return the data in its original class, while `TRUE` will return a list containing the data in its original class and the metadata.

```

# retrieve the frog.data to R's global environment (this does not
# delete it from the library)

extracted.plant.data <- getAMData(x = "plant.data",
  amml = mymodels, as.list = FALSE)

# the extracted data are returned in their original form

class(extracted.plant.data)

[1] "data.frame"

```

As a second example, we will extract the chorus frog data with the `getAMData` function as a list:

```
# retrieve the frog.data to R's global environment as a list containing data and metadata
extracted.frog.data <- getAMData(x = "frog.data", amml = mymodels, as.list = TRUE)
# view the first three records of the extracted.frog.data
lapply(X = extracted.frog.data, FUN = head, n = 3)
$data
Data frame representation of unmarkedFrame object.
      y.1 y.2 y.3 sitevar1 obsvar1.1 obsvar1.2 obsvar1.3
1  1  0  NA  13.02      1          2          3
2  1  0  0   12.92      1          2          3
3  1  0  0   17.75      1          2          3
$metadata
$metadata$comment
[1] "Chorus frog dataset from the package unmarked."
$metadata$taxa
[1] "Chorus Frog"
$metadata$url
[1] "http://www.rdocumentation.org/packages/unmarked/versions/0.11-0/topics/occu"
```

Notice the returned object is a list containing two elements. The first list element is named `data`, which is of class `unmarkedFrameOccu` generated by the package, `unmarked`. The second list element is another list providing the metadata associated with the dataset.

To remove a specific `amData` object from the `amModelLib`, use the `rmData` function. `amData` objects can be deleted by index or by name. Here, we delete the 'sim.data' `amData` object by referencing its current index (4).

```
# remove the amData object, 'sim.data' by index
mymodels <- rmData(amml = mymodels, x = 4)
# notice the sim.data amModel object has been removed, leaving 4 datasets
```

```
lsData(mymodels)
[1] "plant.data"          "updated.plant.data" "apple.data"
[4] "frog.data"
```

Note that all indices have shifted as a result. We can also delete by name. Here, we delete the **amData** 'updated.plant.data':

```
# remove the amModel 'updated.plant.data' from the amModelLib
mymodels <- rmData(amml = mymodels, x = "updated.plant.data")
# notice the updated.plant.data amModel object has been removed,
# leaving 3 datasets
lsData(mymodels)
[1] "plant.data" "apple.data" "frog.data"
```

The Models slot. Models (analysis outputs) are stored in the model slot of the **amModelLib** object, and the model slot is a list of objects of class **amModel**. As with the **amData** class detailed above, the purpose of this class is to store not only the analysis outputs, but also metadata about the outputs. The key functions associated with **amModel** objects (Supplement 1) include:

- **amModel**—creates an object of class **amModel**.
- **insertAMModelLib**—inserts objects of class **amModel** to a **amModelLib** object.
- **rmModel**—deletes an **amModel** object.
- **getAMModel**—extracts an **amModel** object from the **amModelLib** object and returns it to the global environment in its original form.

To illustrate, first we will retrieve our plant data from the library, and then analyze plant weights with the function, **lm** and store the results in an object of class **lm** named **lm.D9**. Next, we will use the **amModel** function to convert the analysis outputs to an **amModel** object (including metadata), and will then add this model to our **amModelLib** with the **insertAMModelLib** function. As with datasets, models must be supplied as a named list and a user can enter any metadata using a key-value syntax.

```
# retrieve the plant data from the library
plant.data <- getAMData(amml = mymodels, x = "plant.data", as.
list = FALSE)
# run the analysis
lm.D9 <- lm(weight ~ group, data = plant.data)
```

```
# create an amModel and add metadata
plant.model <- amModel(model = lm.D9, comment = "Analysis from lm
helpfile",
data = "plant.data")
# insert the model to the amModelLib as a named list
mymodels <- insertAMModelLib(models = list(plant.model = plant.
model), amml = mymodels)
```

You may recall that the **lm.D9** object contains a rich amount of information about the linear model analysis, including the data inputs. The example above adds **lm.D9** to an **amModel** object, and supplies additional metadata about the analysis itself.

In this example, the metadata contains an (optional) element named **data**, which points to the name of the **amData** dataset that was used to create that model. **The relationship established by this pairing is completely informal and no checking is performed to verify the existence or compatibility of the data.** However, accessor functions will look for the keyword "data" in the model's metadata, and will retrieve the dataset that is linked to the model if such a pairing is indicated. The "data" metadata element is currently the only element that will trigger a relationship pairing. [Future versions of AMModels will include a "prior" metadata element that pairs posterior models with their prior models.]

Typing in the name of the **amModel** object will show the object according to the object's original class. Alternately, a user can invoke the `summary` method to look at the **amModel** object, which returns a summary of the model and its associated metadata.

```
# look at the summarized snapshot of an amModel object
summary(plant.model)
Call:
lm(formula = weight ~ group, data = plant.data)
Residuals:
           Min           1Q   Median           3Q          Max
 -1.0710   -0.4938   0.0685   0.2462   1.3690

Coefficients:
           Estimate Std. Error  t value Pr(>|t|)
(Intercept)  5.0320      0.2202  22.850  9.55e-15 ***
groupTrt    -0.3710      0.3114  -1.191   0.249

Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.6964 on 18 degrees of freedom
```

```
Multiple R-squared: 0.07308, Adjusted R-squared: 0.02158
F-statistic: 1.419 on 1 and 18 DF, p-value: 0.249
--- Metadata ---
      name      value
1  comment  Analysis from lm helpfile
2   data    plant.data
3   date    2017-10-04 07:28:29
```

Here, because the model slot in the **amModel** object is occupied by an object of class **lm**, the call is passed to the `summary` method defined for **lm** objects and the results are displayed.

The same steps are used to add the chorus frog occupancy model to the **amModelLib**. Here, we add the results of our two occupancy models (**fm1** and **fm2**) by first creating **amModel** objects with metadata, and then inserting them to the **amModelLib** with the `insertAMModelLib` function.

```
# retrieve the frog data from the library (an unmarkedFrame object)
frog.data <- getAMData(amml = mymodels, x = "frog.data", as.
list = FALSE)

# run the first analysis and store the results as fm1
fm1 <- occu(formula = ~obsvar1 ~ sitevar1, data = frog.data)

# create an amModel object for the first analysis and add metadata
frog.model1 <- amModel(model = fm1, comment = "Occupancy as a func-
tion of sitevar1 and detection as a function of obsvar1.", data =
"frog.data")

# run the second analysis (the intercept only model) and store the
results as fm2
fm2 <- occu(formula = ~1 ~ 1, data = frog.data)

# create an amModel object for the second analysis and add metadata
frog.model2 <- amModel(model = fm2, comment = "Occupancy as a func-
tion of no covariates and detection as a function of no covariates",
data = "frog.data")

# insert both models to the amModelLib as a named list
mymodels <- insertAMModelLib(models = list(frog.model1 = frog.
model1, frog.model2 = frog.model2), amml = mymodels)

# show the contents of the amModelLib object
mymodels
```

Description:

```
[1] This AM Model Library stores analysis inputs (data) and analysis
      outputs (models) associated with the AMModels package
      vignette.
```

Info:

```
owner
[1] Me
email
[1] me@somewhere.com
organization
[1] My Organization
```

Models:

	name	class	package
1	plant.model	lm	<NA>
2	frog.model1	unmarkedFitoccu	unmarked
3	frog.model2	unmarkedFitoccu	unmarked

Data:

	name	class	rows	cols	package
1	plant.data	data.frame	20	2	<NA>
2	apple.data	data.frame	1	2	<NA>
3	frog.data	unmarkedFrameOccu	NA	NA	unmarked

Finally, let's add two Bayesian apple infestation rate models. Recall that this analysis is a user-defined model that has not been formally conducted with a dedicated R package, which would be a normal course of action. Here, we will link the second model to the first model using the 'prior' metadata keyword, and link the second model's data to the 'apple.data' **amData** object using the 'data' metadata keyword.

```
# create apple.m1 as a list
apple.m1 <- list(distribution = "beta", parameters = list
(alpha = 2, beta = 3))
# create an amModel object for apple.m1
apple.m1 <- amModel(model = apple.m1, comment = "Bayes model 1 for
apple infestation")
# create apple.m2 as a list
```

```
apple.m2 <- list(distribution = "beta", posteriors = list
(alpha = 27, beta = 78))

# create an amModel object for apple.m2; link this model to apple.m1
and also associate the dataset.

apple.m2 <- amModel(model = apple.m2, comment = "Bayes model 2 for
apple infestation", prior = "apple.m1", data = "apple.data")

# insert both models to the amModelLib object as a named list

mymodels <- insertAMModelLib(models = list(apple.m1 = apple.m1,
apple.m2 = apple.m2), amml = mymodels)
```

Our **amModelLib** object now contains five **amModel** objects and three **amData** objects.

```
# show the contents of the amModelLib object

mymodels
Description:
[1] This AM Model Library stores analysis inputs (data) and analysis
      outputs (models) associated with the AMModels package
      vignette.
Info:
  owner
    [1] Me
  email
    [1] me@somewhere.com
  organization
    [1] My Organization
Models:
      name      class      package
1 plant.model  lm      <NA>
2 frog.model1  unmarkedFitOccu unmarked
3 frog.model2  unmarkedFitOccu unmarked
4 apple.m1     list     <NA>
5 apple.m2     list     <NA>
```

```
Data:
      name      class      rows cols package
1 plant.data data.frame    20   2    <NA>
2 apple.data data.frame     1   2    <NA>
3 frog.data  unmarkedFrameOccu NA  NA  unmarked
```

The `summary` function would display the same information, but additionally provide the metadata associated with each **amModel** and **amData** object.

Use the `lsModels` function to retrieve the names of the **amModel** objects in an **amModelLib**:

```
# list the amModel objects within the library
lsModels(mymodels)
[1] "plant.model" "frog.model1" "frog.model2" "apple.m1"
"apple.m2"
```

The `modelMeta` function is analogous to the `dataMeta` function and allows metadata to be viewed or set after the model has been inserted into the **amModelLib** object. Although we do not provide an example, it is possible to store the R code used to generate the model as metadata to enhance analytic reproducibility.

To retrieve ("check out") a specific **amModel** object and return its original class to R's global environment, the `getAMModel` accessor function can be used. Let's extract the first frog model with the `getAMModel` function, which requires the name of the **amModel** object. The optional argument, **as.list** indicates how the object is to be returned: `FALSE` (default) will return the model in its original class, while `TRUE` will return a list containing the model in its original class and the metadata.

```
# retrieve frog.model1; this does not remove the model from the
library
extracted.frog.model1 <- getAMModel("frog.model1", amml = mymodels,
as.list = FALSE)
# the extracted model is returned in its original form
class(extracted.frog.model1)
[1] "unmarkedFitOccu"
attr(,"package")
[1] "unmarked"
```


To remove a specific **amModel** object from the **amModelLib**, use `rmModel` function. As with `rmData`, a model can be deleted by index or by name. Here, we delete the **amModel** 'apple.m2':

```
# remove the amModel 'apple.m2' from the amModelLib
mymodels <- rmModel(mymodels, "apple.m2")
# show the amModelLib
mymodels
Description:
[1] This AM Model Library stores analysis inputs (data) and analysis
      outputs (models) associated with the AMModels package
      vignette.
Info:
  owner
  [1] Me
  email
  [1] me@somewhere.com
  organization
  [1] My Organization
Models:
      name      class      package
1 plant.model  lm          <NA>
2 frog.model1  unmarkedFitOccu unmarked
3 frog.model2  unmarkedFitOccu unmarked
4 apple.m1    list        <NA>

Data:
      name      class      rows  cols  package
1 plant.data  data.frame    20    2    <NA>
2 apple.data  data.frame     1    2    <NA>
3 frog.data   unmarkedFrameOccu NA    NA    unmarked
```

Note that all indices have shifted as a result. The shifting indices illustrate a critical concept: *never* link models and data by index number as indices will shift as items are inserted and

removed from the **amModelLib** object. Users who link models with data or models with models should do so by name, and the names should be treated as permanent names to retain the integrity of the library.

Searching and subsetting the amModelLib

Users may create as many **amModelLibs** as needed to help organize a variety of models. As the number of models and datasets grows within an **amModelLib** object, it may be useful to search for objects within it and subset the **amModelLib** object if necessary. The function, `grepAMModelLib` can be used for this purpose; it returns an object of class **amModelLib** containing the matches to the search (if any).

The function uses `grep` to recursively search for a pattern in a given **amModelLib** object; the pattern argument may therefore be a simple string or a regular expression. The user may search both the model and data slots ('all'), or either the model or the data slot by specifying a **search** argument; omitting the **search** argument will default to 'all'.

```
args(grepAMModelLib)
function (pattern, amml, search = c("all", "model", "data"),...)
NULL
```

For example, here we search through all **amData** objects that include the term **frog**.

```
# search for data containing the word 'frog'
grepAMModelLib(pattern = "frog", amml = mymodels, search = "data")
Description:
[1] This AM Model Library stores analysis inputs (data) and analysis
      outputs (models) associated with the AMModels package
      vignette.
Info:
  owner
  [1] Me
  email
  [1] me@somewhere.com
  organization
  [1] My Organization
Models:
      name      class      package
1 frog.model1 unmarkedFitOccu unmarked
2 frog.model2 unmarkedFitOccu unmarked
```

```
Data:
      name      class      rows cols package
1 frog.data unmarkedFrameOccu NA    NA    unmarked
```

The results show an (unsaved) **amModelLib** object which contains an **amData** object called 'frog.data'. Because the frog dataset was linked to frog.model1 and frog.model2 with the "data" metadata element, the **amModelLib** object also contains two **amModel** objects.

Alternatively, the **amModelLib** can be subset with list subsetting. List subsetting methods for **amModelLib** objects also attempt to keep the data with the models if they have been paired by matching against the 'data' metadata element. Here, we subset the library by the name, "plant.model", store the result as 'mymodels2', and verify that the library contains both the model and the linked dataset.

```
# create new amModelLib named plantAnalyses that includes only the
# plant models
plantAnalyses <- mymodels[c("plant.model")]

# update the description
ammlDesc(amml = plantAnalyses) <- "This AM Model Library stores
analysis inputs (data) and analysis outputs (models) associated
with the plant data in the lm helpfile."

# show the new library
plantAnalyses
Description:
[1] This AM Model Library stores analysis inputs (data) and analysis
      outputs (models) associated with the plant data in the lm
      helpfile.

Info:
owner
[1] Me

email
[1] me@somewhere.com

organization
[1] My Organization
```

```
Models:
      name      class package
1 plant.model  lm      NA

Data:
      name      class  rows cols package
1 plant.data data.frame 20   2      NA
```

The `grepAMModelLib` function is intended to subset the **amModelLib** object to allow users to locate datasets and associated models, and vice versa. However, if the subset library is stored, such as in the example above, some models and datasets will be contained in multiple libraries, which could lead to confusion (e.g., the plant analysis is now stored in two libraries, **plantAnalyses** and **mymodels**).

To avoid such duplication, the `grepAMModelLib` function can be used to split an existing **amModelLib**. For example, suppose we decide that the frog analyses should be stored in a separate library. The following code will search for the frog models and datasets, create a new library called "chorusFrogLibrary", and remove them from the original library:

```
# pull out models and data containing the word 'frog' from the
library, mymodels

chorusFrogLibrary <- grepAMModelLib(pattern = "frog",
amml = mymodels, search = "all")

# update the library description

ammlDesc(amml = chorusFrogLibrary) <- "This AM Model Library stores
analysis inputs (data) and analysis outputs (models) associated
with chorus frog research and management."

# remove the amModels now in frog.library from mymodels

mymodels <- rmModel(amml = mymodels, x = lsModels
(chorusFrogLibrary))

# remove the amData now in frog.library from mymodels

mymodels <- rmData(amml = mymodels, x = lsData(chorusFrogLibrary))

# look at the frog.library

chorusFrogLibrary

Description:

[1] This AM Model Library stores analysis inputs (data) and analysis
      outputs (models) associated with chorus frog research and
```

```

management.
Info:
  owner
  [1] Me
  email
  [1] me@somewhere.com
  organization
  [1] My Organization
Models:
      name      class      package
1 frog.model1 unmarkedFitOccu unmarked
2 frog.model2 unmarkedFitOccu unmarked

Data:
      name      class      rows  cols  package
1 frog.data unmarkedFrameOccu NA    NA    unmarked

# look at the original mymodels library; notice the frog analyses
# have been removed, while the plant analyses have been retained
# because they were not removed when subsetting the plant library
mymodels
Description:
[1] This AM Model Library stores analysis inputs (data) and analysis
      outputs (models) associated with the AMModels package
      vignette.
Info:
  owner
  [1] Me
  email
  [1] me@somewhere.com
  organization
  [1] My Organization
Models:
      name      class  package
1 plant.model  lm     NA
2 apple.m1    list   NA

```

```
Data:
      name      class  rows cols package
1 plant.data data.frame 20   2     NA
2 apple.data data.frame  1   2     NA
```

Saving the amModelLib object

You may save your **amModelLib** with either the `save` or `saveRDS` functions from the base package.

```
# save mymodels to an.rda file (which can store multiple objects or multiple libraries)
save(mymodels, file = "mymodels.rda")
# save frog.analyses to an.RDS file (which can store a single library)
saveRDS(chorusFrogLibrary, file = "chorusFrogLibrary.RDS")
```

The file argument should include the filepath where the `.RData` or `.rda` file will live. To call up the **amModelLib**, the `load` or `readRDS` function can be used.

```
# load a stored.rda file
load("mymodels.rda")
# load a stored.rds file (which will store a single object, in this case a single ammodelLib)
chorusFrogLibrary <- readRDS("chorusFrogLibrary.RDS")
```

We envision that saved **amModelLib** objects will be used repeatedly as they contain a storehouse of models and datasets that can be retrieved for future use.

amModelLib organization

Users may maintain multiple **amModelLibs** to best meet their needs. For example, a single **amModelLib** may be used to:

- store models associated with a specific set of analyses, such as those included in an annual report, a scientific paper, or a monitoring program;
- store models associated with a specific parameter of interest (e.g., all models related to apple infestation rate);

- store models associated with a specific species or research project (e.g., the chorus frog analyses);
- store models by model type (all occupancy models);
- store all models by a particular person;
- etc.

Trial and error may reveal the best approach to meet the user's needs. At some point, a library may become quite large and cumbersome, and we hope the tools provided in this package allow users to reorganize **amModelLib** objects as needs change.

AMModels Shiny app

Many natural resource practitioners are time-crunched and may not work with R on a daily basis, facilitating forgetfulness of R workflow and function names. This should not hamper efforts to codify knowledge in the form of models. To that end, we introduce a Shiny application called **AM Model Manager**. To use the application, you must have the shiny and shinyBS packages installed. You can then launch the application with the function `modelMgr` with no arguments. When the application is launched, the R console will be "listening" and unavailable for use.

```
# launch the Model Manager  
modelMgr ()
```

The app enables a user to add, delete, and send **amModel** and **amData** objects from the library to R's global environment, and to work with their associated metadata. Once those tasks are completed, the Model Manager is closed and the R console is then freed for additional work.

The Shiny application can be resized as desired, and is divided into 5 main actions, highlighted in red in [Fig 6](#):

1. Locate, upload, or create an **amModelLib** object.
2. Search or subset the loaded **amModelLib**.
3. Edit a selected **amModelLib** content by adding or removing metadata, models, or data.
4. Bind an **amModelLib** object to R's global environment, where it can be saved with the **save** or **saveRDS** functions within R's console.
5. View the result of actions taken.

Each of these will be briefly demonstrated.

Selecting an amModelLib

Use the blue button at the top of [Fig 6](#) to (1) locate an **amModelLib** that is present in the global environment, (2) create a new library, or (3) upload an .RData, .rda, or .RDS file

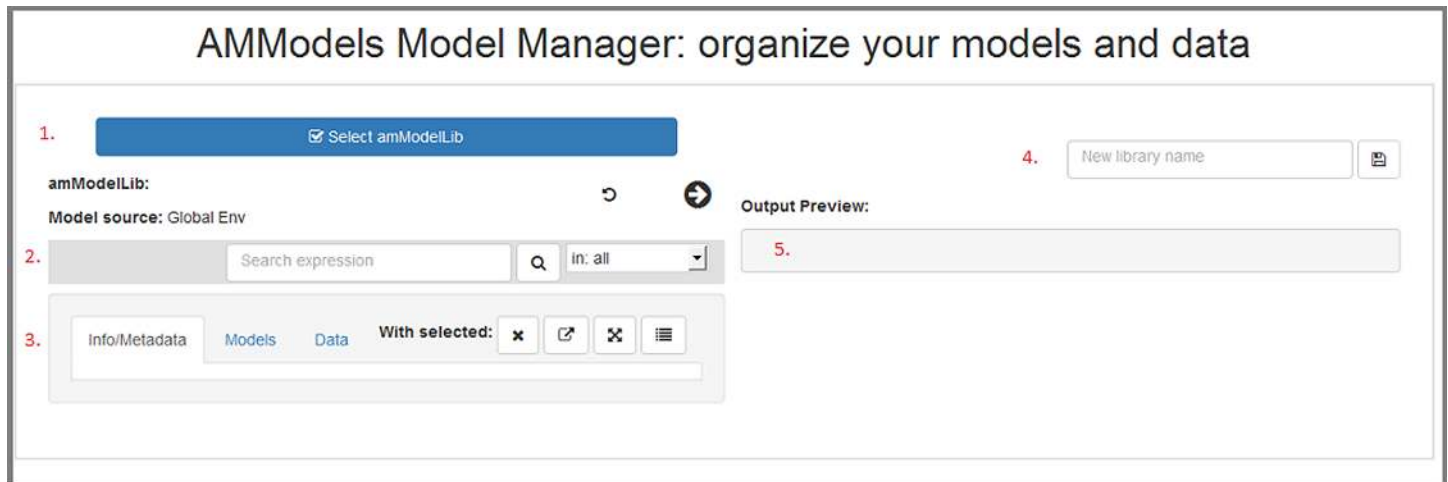


Fig 6. The AMModels Model Manager application main window.

<https://doi.org/10.1371/journal.pone.0188966.g006>

containing one. When the window first opens, the app will list the objects in your global environment, with the filter to display only **amModelLib** objects active (Fig 7). An **amModelLib** object may be selected from the list displayed if desired. In Fig 7, the **chorusFrogLibrary** has been selected.

Alternatively, an **amModelLib** can be created by pressing the "New" button in the app (Fig 7). Creating a new **amModelLib** launches another dialogue box on top of the first one where the user can specify the name of the new **amModelLib**, provide a description, and begin entering metadata associated with the library itself (e.g., Fig 8). The name must be a valid R object name that does not start with a number or contain a space. If these are ignored, the name will be coerced to a valid name using the `make.names` functions. The "Create **amModelLib**" button at the bottom of the dialogue box adds the **amModelLib** object to the global environment, loads it as the **amModelLib** chosen for editing in the Model Manager app, and closes the dialogue box. To cancel, choose "Close" at the lower right, or click anywhere outside of the dialogue box.

Finally, a user may press the Browse button to browse to an existing **.RData** or **.rda** file containing one or more **amModelLib** objects, or an **.RDS** file containing a single **amModelLib** object (Fig 9). For **.RData** and **.rda** files, all objects within the file are loaded into the global environment and displayed as a list in the dialogue box where the user may select one. For **.RDS** files, the **amModelLib** object is loaded into the global environment using the basename of the uploaded file, minus the **.RDS** extension. In Fig 9, we browse to the **amModelLib** called **chorusFrogLibrary.RDS** that was previously created in this vignette.

Editing contents of a loaded library

All edits are contained in the app itself; ultimately any edits will need to be saved as a new object (library) to R's global environment, where they can be saved to file when the app has been closed. Pressing the right-pointing arrow that is highlighted in red will display a summary of current contents of the loaded library (Fig 10). Here, we discuss how edits within the app are made.

Info/Metadata tab. The info/metadata tab contains information about the **amModelLib** itself (Fig 10). The description field is a place to store notes about the contents of the **amModelLib**, while the metadata consist of name-value pairs. Each name-value pair is entered by

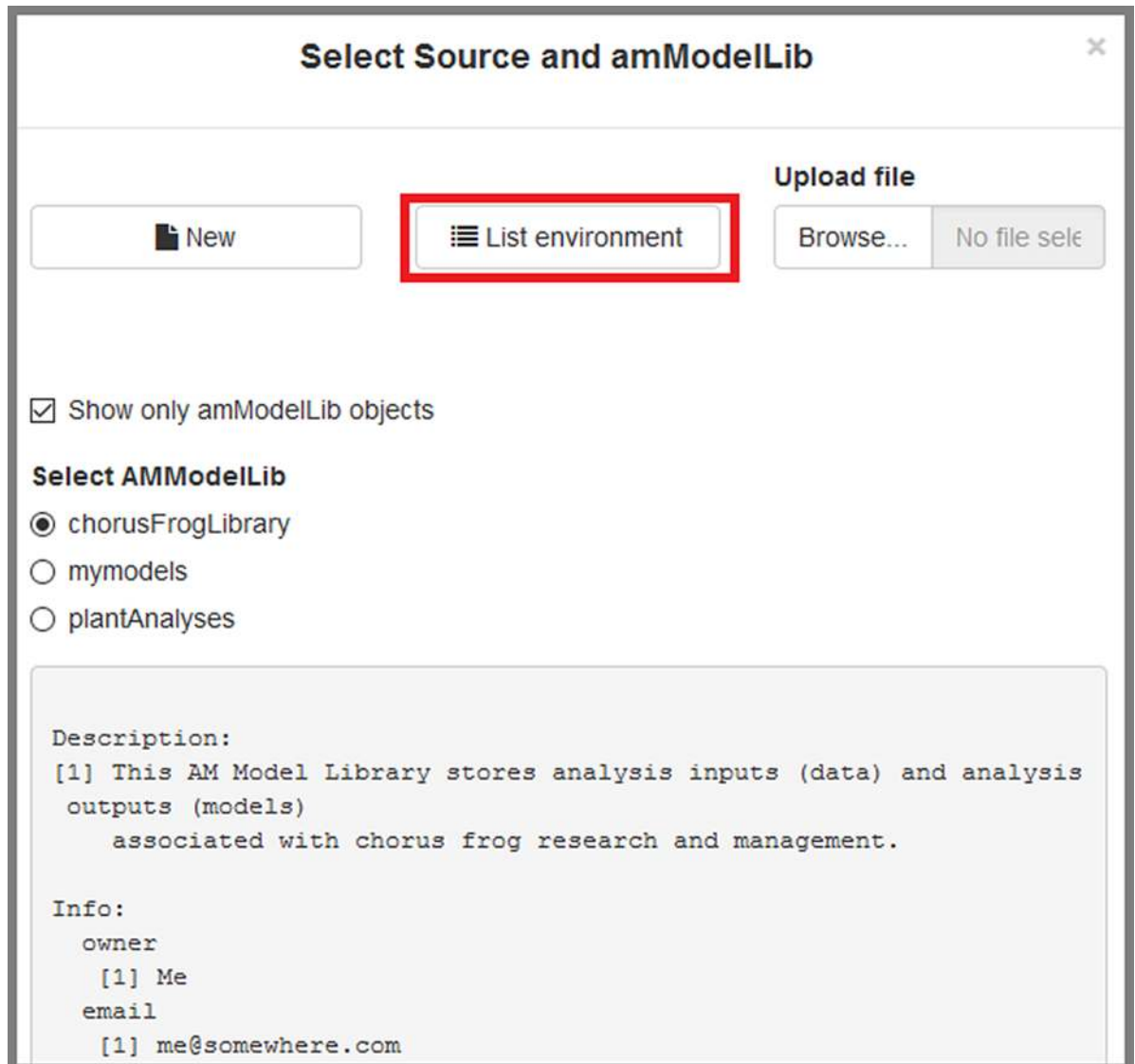


Fig 7. The "Select amModelLib" window allows users to select a library that exists in R's global environment.

<https://doi.org/10.1371/journal.pone.0188966.g007>

typing a name for each item and its corresponding value. The values are text fields of any length. After entering a name-value pair, press the "thumbs-up" button to the left of the row to add the item and create an empty row at the bottom of the table, where another pair may be added. To remove a name-value pair, check the box to the left of the row and press the thumbs-up button.

Models tab. The models tab lists the models (analysis outputs) in the **amModelLib** (Fig 11). The "Summary" button in each row opens a dialogue box that displays a summary of the model in that row, invoking the summary method of the object's class. Similarly, the "Edit" button in each row opens a dialogue box that allows the user to edit the metadata associated with the model of that row.

Additional models may be added by clicking the "Add model" button, which opens a dialogue box in which objects from the global environment may be selected for inclusion. The

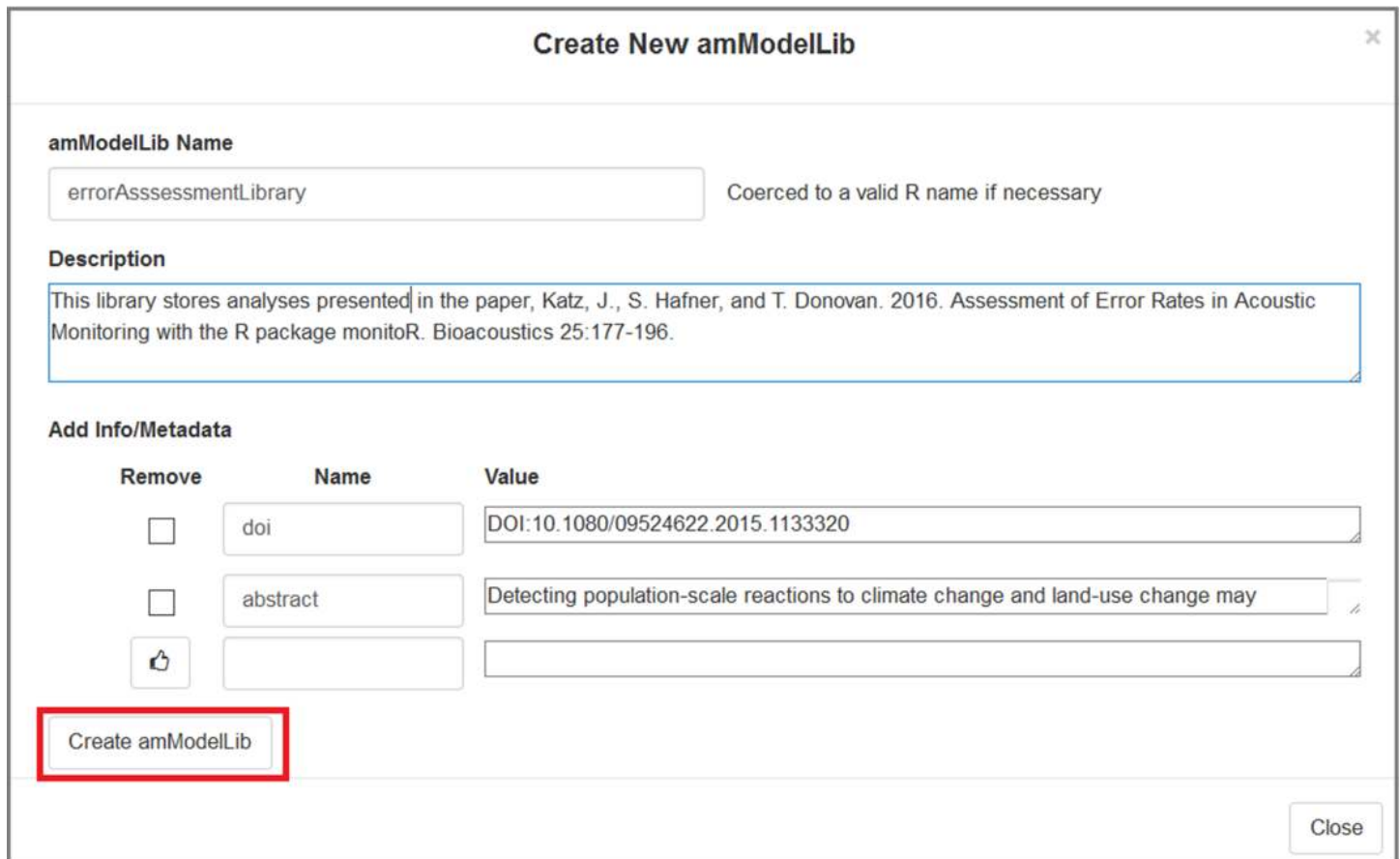


Fig 8. The "Create New amModelLib" window allows users to create a new library.

<https://doi.org/10.1371/journal.pone.0188966.g008>

action will convert the object into an **amModel** object and add it to the library. Once the new model has been added, the user can return to the models tab and may press the "Edit" button to supply metadata.

One or more models may be selected by checking the box to the left of the name. Alternatively, all models may be selected or de-selected by pressing the button above the word "Select." Four actions may be taken with selected items, which will be described shortly.

Data tab. The data tab lists the data in the **amModelLib**. Layout and function of the data tab is identical to the models tab (Fig 12).

Actions available for selected components. Four actions can be performed with selected components (Fig 13).

When an action is chosen it is performed on all selected components, both models and data, even though only one of the models or data tab can be visible at a time. Actions are selected using the buttons to the right of the tab-selection bar.

1. Delete. The "delete" action removes the selected components from the loaded **amModelLib** object.
2. Create new **amModelLib**. This action sets the selected objects as a new, unsaved **amModelLib** object. Pressing the right arrow button allows the user to see a summary of this new library.

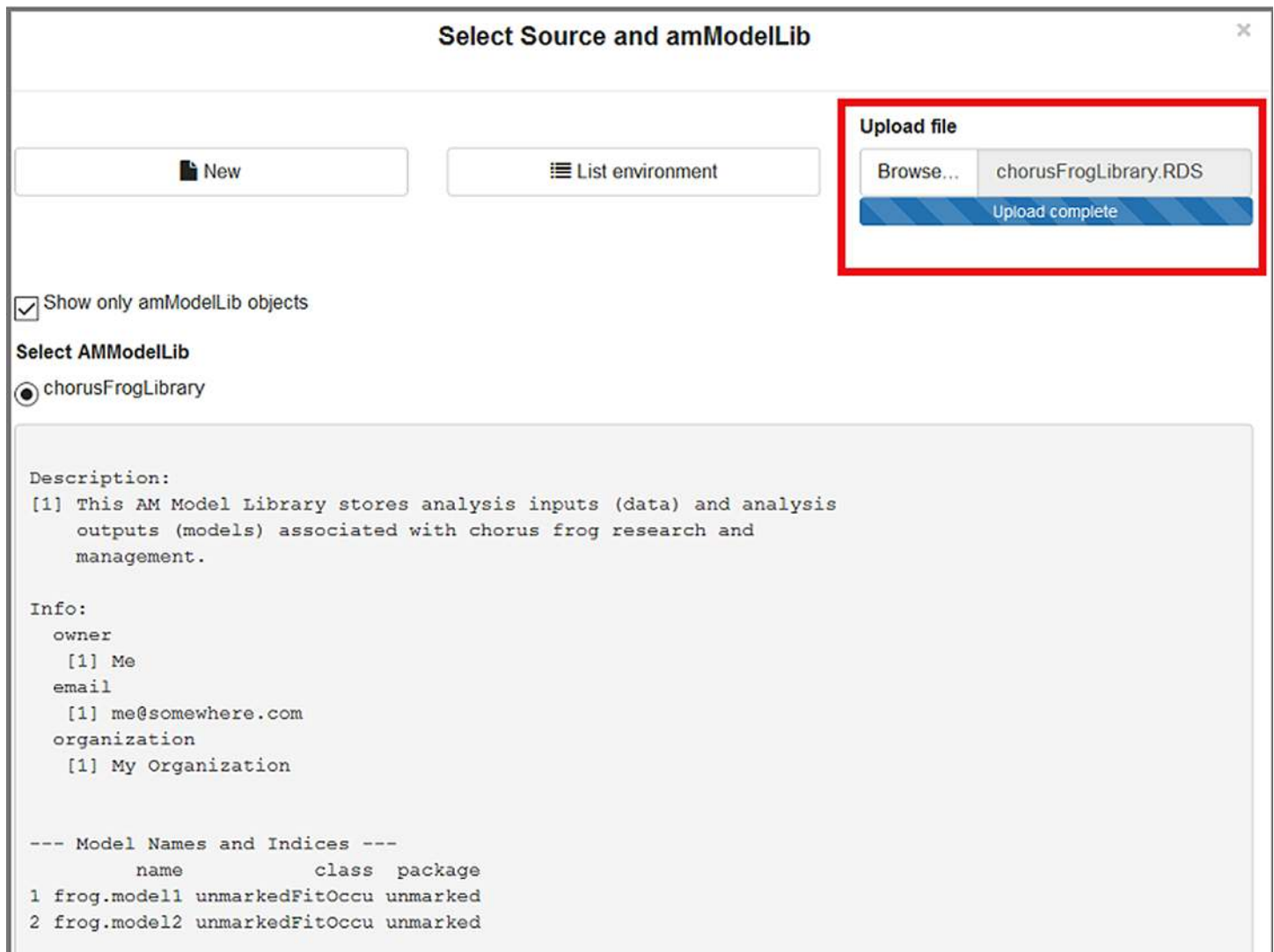


Fig 9. The "Upload file" window allows users to browse to an existing library that is stored as an.RDS or.rda file.

<https://doi.org/10.1371/journal.pone.0188966.g009>

3. Extract ("check out") an original object from the library to the global environment without metadata (uses `getAMModel` or `getAMData`)
4. Extract ("check out") as a list the original object plus the metadata to the global environment (uses `getAMModel` or `getAMData` with the argument `as.list = TRUE`). This extraction action, and the one above, immediately place the selected objects into the global environment. Each object placed in the global environment is assigned the same name as in the `amModelLib` object, replacing existing objects with the same name without warning.

For the first two actions, pressing the right-arrow button will show a summary of the resulting `amModelLib`. (Extracting objects will not change the library.)

To undo all actions since the `amModelLib` was loaded, e.g. to undo accidentally deleting some components, click the "revert" button above the search bar (the counter-clockwise arrow).

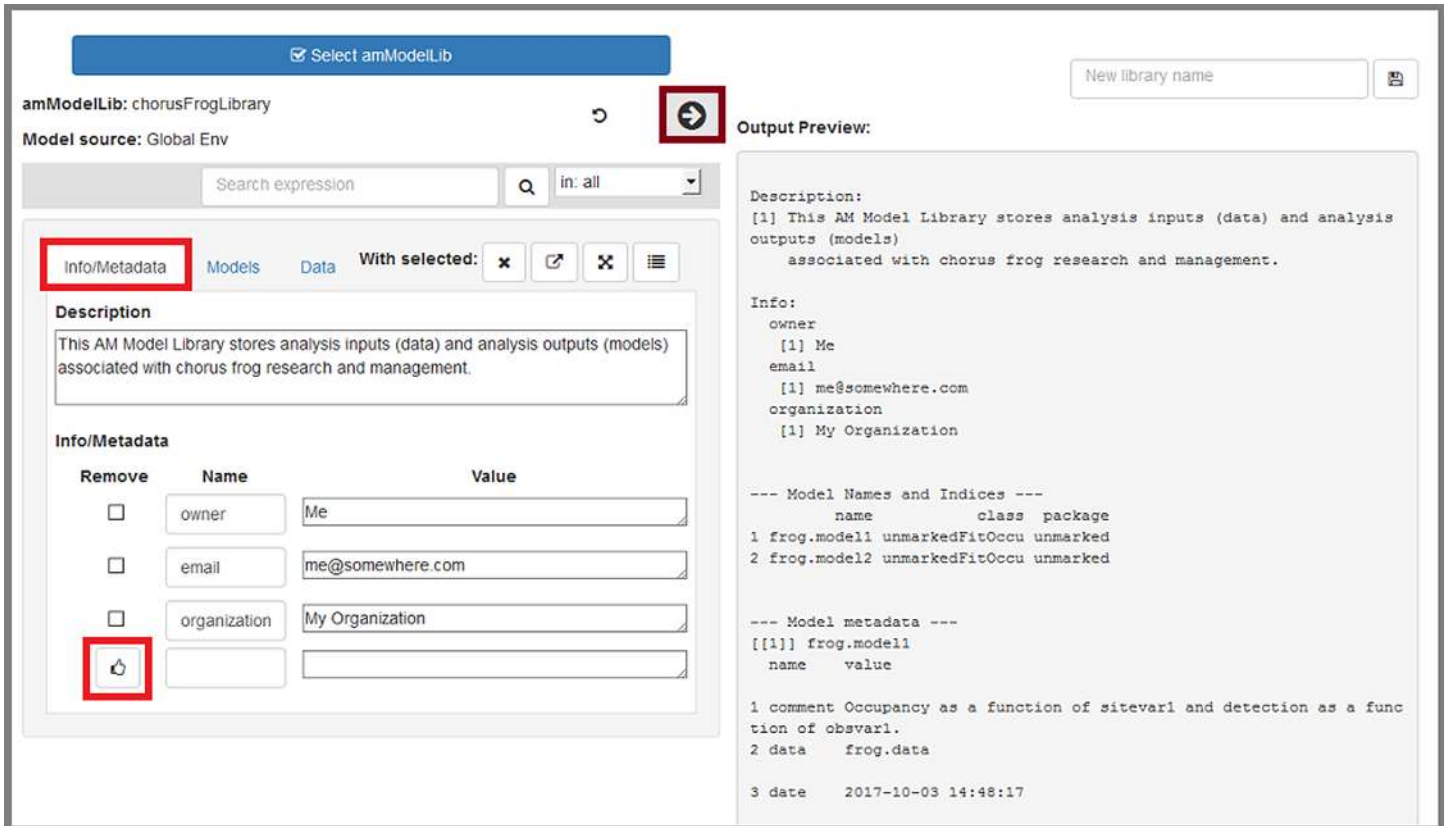


Fig 10. The "Info/Metadata" tab allows users to work with the library's metadata and info slots.

<https://doi.org/10.1371/journal.pone.0188966.g010>

Searching for content

The search box allows users to search the models or datasets in the **amModelLib** using the `grepAMModelLib` function. Enter a search term or expression in the box, choose whether to search all models and data, just models, or just data, and click the magnifying glass button (Fig 14). A summary of the result of the search is displayed in the output preview pane when the right arrow button is pressed. The results are contained in a new **amModelLib** within the app itself.

Saving changes

All edits are contained in the app itself; ultimately any edits will need to be assigned to a new object (library) and moved to R's global environment, from where it can be saved to a file using either the `save` or `saveRDS` functions in the R console.

To save **all** edits to the loaded library, press the right-arrow to preview the updated contents of the loaded library. Next, supply a "New library name" at the top of the output preview pane and press the "bind" button (with a disk icon which looks like a save button). This action places the library in R's global environment. If the name provided currently exists in the global environment, the original library will be replaced with the updated library. If this action is successfully executed, a success indicator will flash on the screen. Then, once the app is closed, the library within the global environment can be saved with the `save` or `saveRDS` functions.

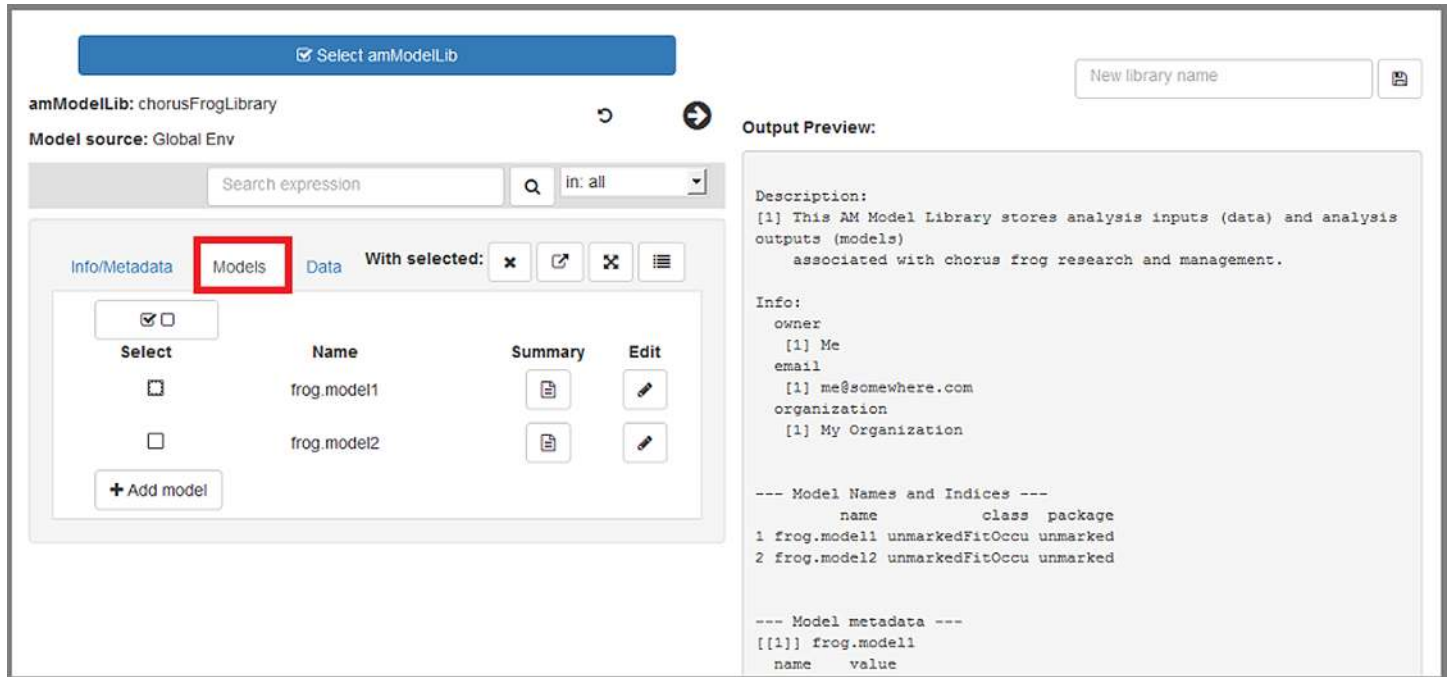


Fig 11. The "Models" tab allows users to add new models, delete existing models, or send models to R's global environment.

<https://doi.org/10.1371/journal.pone.0188966.g011>

To create a **new** library that is a subset of objects in the loaded library, select the models and data of interest, preview the results of the library subset by pressing the right arrow button, then provide a unique "New library name" and press the "bind" button. If this action is successfully executed, a success indicator will flash on the screen. Once the library is placed in the global environment, the app can be closed and the new **amModelLib** object can be saved to a file using either the `save` or `saveRDS` functions. Be cautious of maintaining items that are stored in multiple libraries.

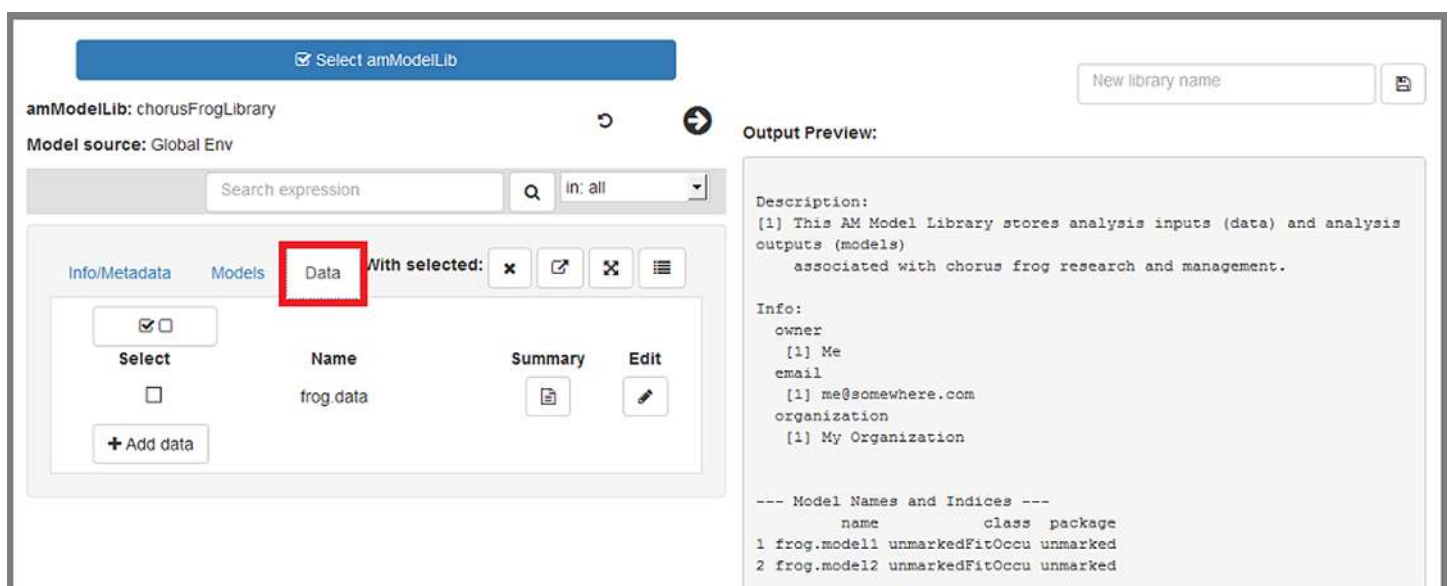


Fig 12. The "Data" tab allows users to add new datasets, delete existing datasets, or send datasets to R's global environment.

<https://doi.org/10.1371/journal.pone.0188966.g012>

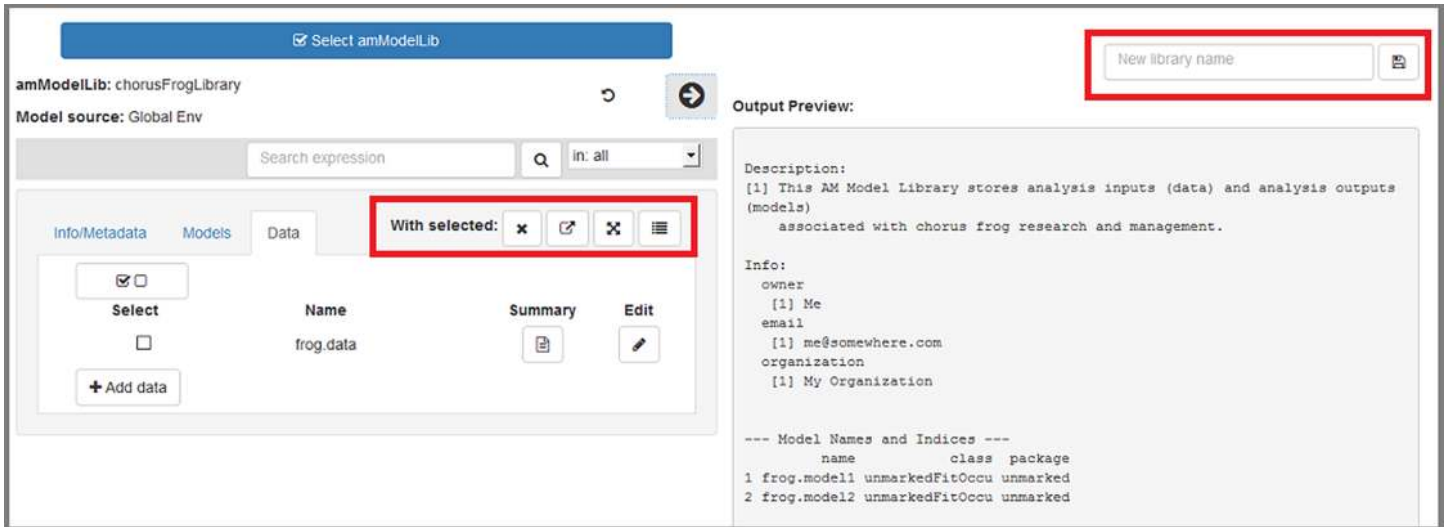


Fig 13. Models and data that are selected can be deleted, subset to a new library, or sent to R's global environment in its original form or as a list.

<https://doi.org/10.1371/journal.pone.0188966.g013>

Versioning

AMModels was developed under R version 3.2.2. We made an attempt to eliminate unnecessary dependencies and imports to minimize the chance that a dependent package update will

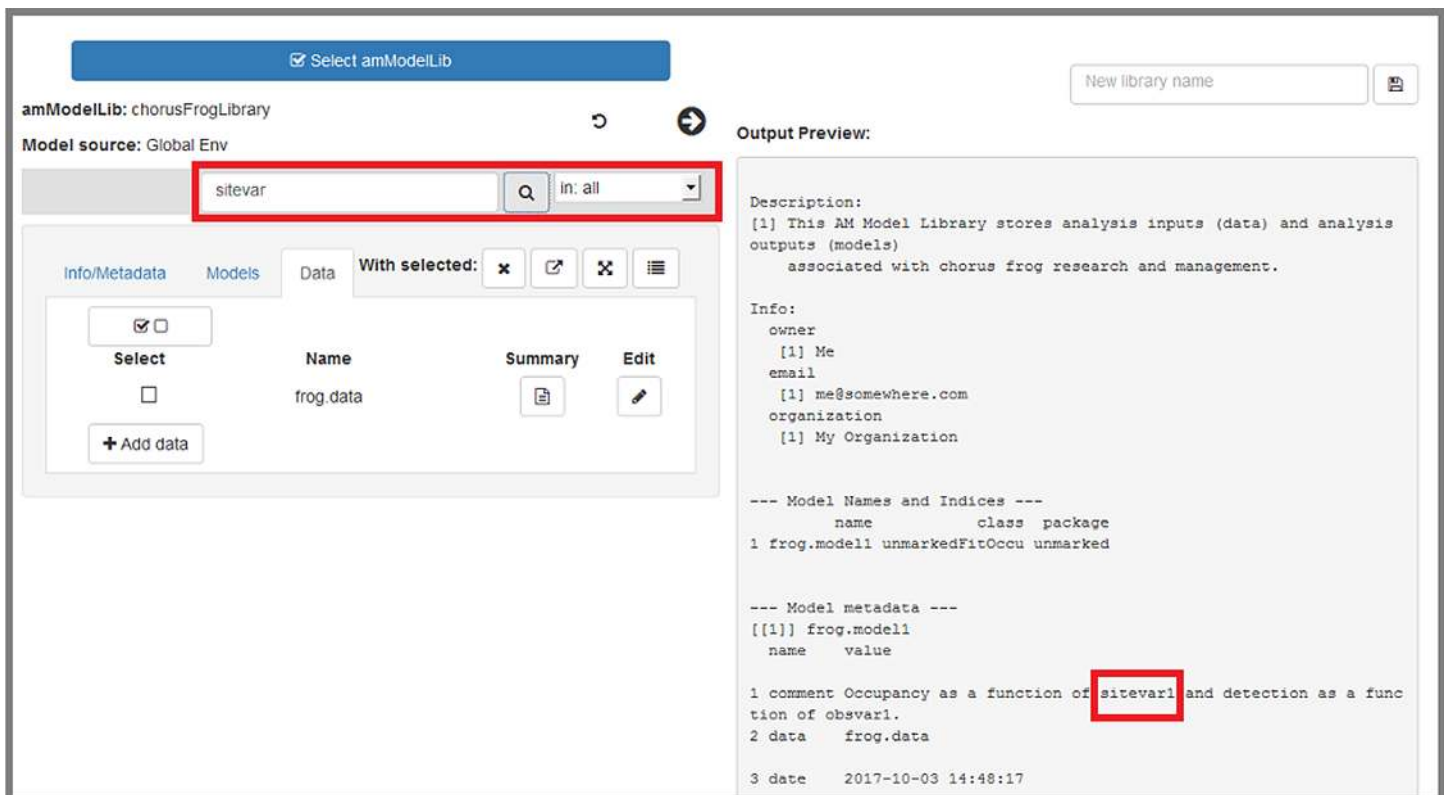


Fig 14. The "Search" panel allows users to search a library by term. Pressing the right arrow will display the result, which can be saved as a new library by providing a name in the New object name box and pressing the Save button, which will bind this new library to R's global environment, where the library can be saved with the save or saveRDS function.

<https://doi.org/10.1371/journal.pone.0188966.g014>

break functions in this package. The shiny user interface was developed under shiny 1.0.0; this interface is completely dependent on the shiny package, and we expect that we will need to update the shiny UI regularly due to the fast pace of shiny package development. Since the premise of our package is long-term storage, we intend to retain backwards compatibility in our updates to the greatest extent possible, short of asking users to stop updating R. We cannot predict which datasets or analytical routines may be rendered incompatible with a new major version of R or a new package version, so we recommend users note R and package versions in their metadata to retain the information necessary to reproduce an analysis long into the future.

Summary

Hallgren and Westberg [33] note, "Behind [the theory of AM] lies a strong, albeit implicit, expectation that organizations aiming for AM have the capacity to communicate in a way that facilitates the required coordination of the knowledge perspectives involved." Thus, at an agency or organizational level, adaptive management requires knowledge management—defined as the process of capturing, developing, sharing, and effectively using organizational knowledge to fulfill its mission [34]. Models are a main ingredient of adaptive management programs, and are a formal representation of knowledge. They play a key role in representing uncertainty, can be compared with competing models, and can be used to predict the outcome of a given management action [35]. The R package, **AMModels**, is a flexible and simple package intended to facilitate adaptive management efforts by storing models, along with datasets, so that they can be used to aid in decision making. Although an adaptive management framework includes many other critical ingredients [11,36,37], **AMModels** may provide a useful tool in advancing the use of adaptive management.

The utility of **AMModels**, however, may be useful to those not engaged in an adaptive management setting. Preserving data, outputs, and metadata (that may include the analytical code) can promote reproducibility of scientific analysis [38–40] and foster the use of models into the future.

Supporting information

S1 Fig. Function cheat sheet for the R package, AMModels.
(PDF)

S1 Text. R script that used in the AMModels examples.
(TXT)

Acknowledgments

We thank Sasha Hafner and Andy Royle for their review of the AMModels package and this article. We also thank Cathleen Balantic and two anonymous reviewers for their thoughtful comments and suggestions. We thank Mark Massar for the opportunity to consider what tools are needed to implement a large scale wildlife monitoring program within an adaptive framework. Use of trade names or products does not constitute endorsement by the U.S. Government. The Vermont Cooperative Fish and Wildlife Research Unit is jointly supported by the U.S. Geological Survey, University of Vermont, Vermont Department of Fish and Wildlife, and Wildlife Management Institute.

Author Contributions

Conceptualization: Therese M. Donovan, Jonathan E. Katz.

Methodology: Therese M. Donovan, Jonathan E. Katz.

Project administration: Therese M. Donovan.

Software: Therese M. Donovan, Jonathan E. Katz.

Supervision: Therese M. Donovan.

Validation: Therese M. Donovan, Jonathan E. Katz.

Writing – original draft: Therese M. Donovan, Jonathan E. Katz.

Writing – review & editing: Therese M. Donovan, Jonathan E. Katz.

References

1. Katz J, Donovan T. AMModels: Adaptive management model manager. Comprehensive R Archive Network; 2017. Available: <https://cran.r-project.org/web/packages/AMModels/>
2. Holling CS, United Nations Environment Programme. Adaptive environmental assessment and management. Laxenburg, Austria; Chichester, New York: International Institute for Applied Systems Analysis; Wiley; 1978. pp. xviii, 377p.
3. Walters C. Adaptive management of renewable resources. New York: Macmillan; 1986. p. 374 p.
4. Lee K. Compass and gyroscope: Integrating science and politics for the environment. Washington DC: Island Press; 1993. p. 255 p.
5. Allen CR, Garmestani AS, editors. Adaptive management of social-ecological systems. Springer Science Mathplus Business Media; 2015. <https://doi.org/10.1007/978-94-017-9682-8>
6. Williams BK, Brown ED. Adaptive management: The u.S. department of interior applications guide. U. S. Department of the Interior, Washington, DC. 2012.
7. Gregory R, Ohlson D, Arvai J. Deconstructing adaptive management: Criteria for applications to environmental management. Ecological Applications. 2006; 16: 2411–2425. PMID: [17205914](https://pubmed.ncbi.nlm.nih.gov/17205914/)
8. Rehme SE, Powell LA, Allen CR. Multimodel inference and adaptive management. Journal of Environmental Management. 2011; 92: 1360–1364. <https://doi.org/10.1016/j.jenvman.2010.10.012> PMID: [20961682](https://pubmed.ncbi.nlm.nih.gov/20961682/)
9. Fontaine JJ. Improving our legacy: Incorporation of adaptive management into state wildlife action plans. Journal of Environmental Management. 2011; 92: 1403–1408. <https://doi.org/10.1016/j.jenvman.2010.10.015> PMID: [20971548](https://pubmed.ncbi.nlm.nih.gov/20971548/)
10. Greig LA, Marmorek DR, Murray C, Robinson DCE. Insight into enabling adaptive management. Ecology and Society. 2013; 18.
11. Rist L, Felton A, Samuelsson L, Sandstrom C, Rosvall O. A new paradigm for adaptive management. Ecology and Society. 2013; 18.
12. Fischman RL, Ruhl JB. Judging adaptive management practices of us agencies. Conservation Biology. 2016; 30: 268–275. <https://doi.org/10.1111/cobi.12616> PMID: [26306648](https://pubmed.ncbi.nlm.nih.gov/26306648/)
13. Williams BK, Brown ED. Technical challenges in the application of adaptive management. Biological Conservation. 2016; 195: 255–263.
14. Runge MC. An introduction to adaptive management for threatened and endangered species. Journal of Fish and Wildlife Management. 2011; 2: 220–233.
15. Fuller AK. Decision making in natural resource management: A structured adaptive approach. Journal of Wildlife Management. 2014; 78: 175–176.
16. R Core Team. R: A language and environment for statistical computing. Vienna, Austria: R Foundation for Statistical Computing; 2016. Available: <https://www.R-project.org/>
17. Balantic C, Katz J, Donovan T. AMMonitor: Remote monitoring of wildlife in an adaptive framework. R package pending submission to CRAN; in prep.
18. Katz J, Donovan T. AMExpert: Deriving models via expert opinion. R package pending submission to CRAN; in prep.
19. MacKenzie D, Nichols J, Royle J, Pollock K, Bailey L, Hines J. Occupancy estimation and modeling: Inferring patterns and dynamics of species. Amsterdam; Boston: Elsevier; 2002. p. 344 p.

20. Fiske IJ, Chandler RB. Unmarked: An R package for fitting hierarchical models of wildlife occurrence and abundance. *Journal of Statistical Software*. 2011; 43: 1–23.
21. Weir LA, Mossman MJ. North american amphibian monitoring program (NAAMP). The conservation status of united states species. University of California Press; 2005. pp. 307–313. <https://doi.org/10.1525/california/9780520235922.003.0045>
22. Burnham KP, Anderson DR. Model selection and inference. Springer New York; 1998. <https://doi.org/10.1007/978-1-4757-2917-7>
23. Mazerolle MJ. AICcmodavg: Model selection and multimodel inference based on (q)AIC(c). 2016. Available: <http://CRAN.R-project.org/package=AICcmodavg>
24. Plummer M, Stukalov A, Denwood M. Rjags: Bayesian graphical models using mcmc. 2016. Available: <http://mcmc-jags.sourceforge.net>
25. Burnetas AN, Katehakis MN. Optimal adaptive policies for markov decision processes. *Mathematics of Operations Research*. 1997; 22: 222–255.
26. Bolker B. Ecological models and data in r. Princeton University Press; 2008.
27. Jaksch T, Ortner R, Auer P. Near-optimal regret bounds for reinforcement learning. *Journal of Machine Learning Research*. 2010; 11: 1563–1600.
28. Conroy M, Peterson J. Decision making in natural resource management: A structured, adaptive approach. Wiley-Blackwell; 2013. Available: <http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470671742.html>
29. Goodwin P, Wright G. Decision analysis for management judgment. John Wiley & Sons; 2014.
30. Hobbs NT, Hooten MB. Bayesian models. Princeton University Press; 2015. <https://doi.org/10.1515/9781400866557>
31. Royle JA, Kery M. Applied hierarchical modeling in ecology. Elsevier; 2016. <https://doi.org/10.1016/c2013-0-19160-x>
32. Chang W, Cheng J, Allaire J, Xie Y, McPherson J. Shiny: Web application framework for R. 2016. Available: <http://CRAN.R-project.org/package=shiny>
33. Hallgren L, Westberg L. Adaptive management? Observations of knowledge coordination in the communication practice of swedish game management. *Wildlife Biology*. 2015; 21: 165–174.
34. Davenport TH. Saving it's soul: Human centered information management. *Harvest Business Review*. 1994; 72: 119–131.
35. Williams BK. Adaptive management of natural resources-framework and issues. *Journal of Environmental Management*. 2011; 92: 1346–1353. <https://doi.org/10.1016/j.jenvman.2010.10.041> PMID: [21075505](https://pubmed.ncbi.nlm.nih.gov/21075505/)
36. Susskind L, Camacho AE, Schenk T. A critical assessment of collaborative adaptive management in practice. *Journal of Applied Ecology*. 2012; 49: 47–51.
37. Westgate MJ, Likens GE, Lindenmayer DB. Adaptive management of biological systems: A review. *Biological Conservation*. 2013; 158: 128–139.
38. Merali Z. Computational science: Error. why scientific programming does not compute. *Nature*. 2010; 467: 775–777. <https://doi.org/10.1038/467775a> PMID: [20944712](https://pubmed.ncbi.nlm.nih.gov/20944712/)
39. Miller G. A scientist's nightmare: Software problem leads to five retractions. *Science*. 2006; 314: 1856–1857. <https://doi.org/10.1126/science.314.5807.1856> PMID: [17185570](https://pubmed.ncbi.nlm.nih.gov/17185570/)
40. Peng RD. Reproducible research in computational science. *Science*. 2011; 334: 1226–1227. <https://doi.org/10.1126/science.1213847> PMID: [22144613](https://pubmed.ncbi.nlm.nih.gov/22144613/)