Modeling distributions of carnivores in western Minnesota and northwestern Iowa

Glen A. Sargeant & Marsha A. Sovada
USGS Northern Prairie Wildlife Research Center
gsargeant@usgs.gov; msovada@usgs.gov

September 2007

Contents

List of Figures

List of Tables

1 Introduction

Accurate maps of species ranges are among the most fundamental information needs for wildlife conservation and management. Ranges of species, however, are surprisingly difficult to estimate. Censuses of large geographic areas are rarely practical; species of interest often are difficult to detect because they are cryptic, secretive, or occur at low density; and many species are generalists or are constrained by other factors to a subset of suitable habitat. These issues motivate interest in methods that support spatially explicit predictions for sites that have not been surveyed, produce valid estimates even when detection is uncertain, and do not necessarily rely on habitat covariates for prediction.

Sargeant et al. (2005) derived such a method when they extended the fully Bayesian Markov random-field model of Heikkinen and Högmander (1994) to accommodate uncertain detection by incorporating a geometric model for detection histories. Their approach permitted estimation from cursory searches of sample map units and thus facilitated economical surveys of large geographic areas. Data collection for their swift fox survey, for example, encompassed approximately 66,000 km^2 for a total cost of $21,945.

Although the model derived by Sargeant et al. (2005) resolved several key practical problems associated with range estimation, it can be difficult to implement with statistical software. In this report, 1) we adapt the model presented by Sargeant et al. (2005) for application to carnivore surveys conducted in western Minnesota and northwestern Iowa during 2003-2004 and 2) document and demonstrate methods we used for data processing, model fitting, model checking, and presentation of results. Biological interpretations of results are presented in a companion report (Sovada and Sargeant 2007).

2 Software

2.1 The R language and environment

We used the R Language and Environment (R Development Core Team 2005) to implement analyses described in this report. Version 2.5 is included on the CD containing this report and can also be obtained for free via the internet (http://www.r-project.org/). Detailed instructions for installation are included as an appendix.

We packaged our analysis as a suite of R *functions*: executable objects that automate and simplify repetitive tasks. Novice R users can thus repeat steps in our analysis, or apply the same procedures to new data, by supplying simple commands that consist of *assignments* and *function calls*. Function calls consist of function names, followed, in parentheses, by *arguments*. In the example below, `ReadGridDef()` names a function. Arguments include *`shapefile`*, *`IDvar`*, and *`centroids`*, which supply necessary information and control behavior of the function. Output produced by the function call is assigned to an *object* called `MN.grid` and thereby preserved for reuse.

```
> MN.grid = ReadGridDef(shapefile = "MNgrid", IDvar = "POLY",
+     centroids = T)
```

Our functions require few arguments, so they are easy to understand and implement. This simplicity limits flexibility, but reduces demands on novice R users. A sufficient understanding of R can be acquired by reading any introduction to R fundamentals, e.g., chapter 1 of *Introductory statistics with R* by Peter Dalgaard (Springer, 2001; http://www.biostat.ku.dk/~pd/ISwR.html).

R help
R includes a full set of manuals as well as an HTML help file for each function. These can be accessed via the "Help" menu or via the command line. For example, commands shown below (1) call the help file for a function (*`foo`*), (2) start the HTML help utility , and (3) return a list of functions related to *`keyword`*.

```
>?foo
>help.start()
>help.search("keyword")
```

2.2 R packages

Literally hundreds of user-contributed "packages" extend the functionality of R. Some of these are included with the base distribution: others can be downloaded from CRAN (the **C**omprehensive **R** **A**rchives **N**etwork; http://cran.r-project.org/) via the internet.

The CD containing this report includes 5 packages, which are stored as `.zip` files in the `Packages` subdirectory and described below. Examples in this report require 4 of the packages: `chron`, `FFWMDfunctions`, `maptools`, and `sp`.

`chron`	A package written by David James and Kurt Hornik; supports operations with dates and times; available from CRAN.
`FFWMDdata`	A package written by the authors of this report; contains datasets and documentation for carnivore surveys conducted by the U.S. Geological Survey and U.S. Fish and Wildlife Service during 2004-2005.
`FFWMDfunctions`	A package written by the authors of this report; functions described in this report and associated documentation.
`maptools`	A package written by Nicholas J. Lewin-Koh and Roger Bivand; tools for reading and manipulating spatial data; available from CRAN.
`sp`	A package written by Edzer J. Pebesma and Roger Bivand; classes and methods for spatial data; available from CRAN.

Packages must be installed before they can be used . The pull-down "Packages" menu includes self-explanatory utilities for installing packages either from CRAN or from the local `.zip` files included with this report on CD.

After installation, packages must be loaded at the start of each R session. Loading can be accomplished either with the pull-down "Packages" menu or as below, with the R function `library()`. Note that loading `FFWMDfunctions` will automatically load `chron` and `maptools`, and `maptools` will automatically load `sp`, so only one command is required.

```
> library(FFWMDfunctions)
```

Functions included in `FFWMDfunctions` and used to prepare this report are summarized

below. `FitModel()` is currently useful only for the sampling design described in subsequent examples. Future revisions are likely and will be undertaken with the objective of enhancing flexibility and the potential for use with other datasets.

Table 1: *Summary of functions in package* ***FFWMDfunctions****.*

Function	**Description**
`ReadGridDef()`	Currently an alias for `ReadShapePoly()`; imports a description of the survey region
`GetNList()`	Identifies neighbors for each polygon in the survey region
`PlotPolys()`	Facilitates preparation of complex graphics with `plot()`; produces color-coded maps of the survey region
`PlotN()`	Facilitates preparation of complex graphics with `plot()`; produces maps that facilitate inspections of neighborhoods
`FormatData()`	Imports and formats survey records
`MakeHist()`	Compiles search and detection histories
`AddPolyData()`	Merges data with a description of the survey region
`InputList()`	Creates a list containing input required for model fitting
`FitModel()`	Executes the model fitting algorithm
`Convergence()`	Facilitates preparation of complex graphics with `plot()`; used to inspect convergence of parameter estimates
`Acceptance()`	Computes acceptance rate for proposals
`Normality()`	Facilitates use of `qqnorm()`; prepares normal probability plots for model parameter estimates
`Distribution()`	Facilitates use of `density` and `plot` to prepare kernel density plots for model parameters
`PlotSampleProbs()`	Distribution and convergence plots for sample polygons
`ChainStats()`	Extracts estimates of parameters from model output.
`GetIndex()`	Extracts estimated probabilities of occurrence from results of model fitting
`DrawMap()`	Facilitates mapping of index values with `plot()`.

3 Describing the survey region

As a first step, we will import and format the description of a survey region, identify neighbors for each polygon in the survey region, and inspect neighborhoods for errors.

The survey region we will use encompassed 50,690 km^2 in western Minnesota and northwestern Iowa, and was partitioned into a semi-regular grid of 1,226 polygons of approximately 41.4 km^2. Our description of the region is stored in an ESRI shapefile called `MNgrid`. Component files include `MNgrid.shx`, `MNgrid.shp`, `MNgrid.sbx`, `MNgrid.sbn`, and `MNgrid.dbf`. Subsequent examples will require this grid definition to be stored in the working directory used for analyses. The pulldown File menu can be used to change the working directory within an R session.

We will use `ReadGridDef()` to import the description of our survey region. Required arguments include the name of the source shapefile (e.g., `MNgrid`) and the name of an ID variable in `MNgrid.dbf` (e.g., `POLY`) that uniquely labels each polygon in the shapefile. If the ID variable includes duplicates or missing values, `ReadGridDef()` will fail and issue a warning.

`ReadGridDef()` computes the centroid for each polygon by default, but this calculation can be suppressed by supplying an optional argument (`centroids=F`; `F` is short for `FALSE`). For simplicity and consistency with the convention we will follow when naming variables within dataframes, `ReadGridDef()` automatically converts variable names to lower case.

```
> MN.grid = ReadGridDef(shapefile = "MNgrid", IDvar = "POLY")
```

> R is case sensitive: for example, *POLY* $\neq$ *poly*. If variables in input files are upper- or mixed-case, they will be converted to lower case by `ReadGridDef()`.

Objects produced by `ReadGridDef()` (e.g., `MN.grid`) are of class SpatialPolygonsDataFrame. As the name implies, objects of this class have 2 primary components. The first is an object of class dataframe (generated from `MNgrid.dbf`) with one record for each polygon in the survey region. The second is an object of class polygons, which is a list of polygon definitions.

Everything used in R is an object, and every object belongs to a class. Classes distinguish objects that contain the same types of information stored in a consistent format. Classes are important because many functions operate only on certain classes. Others, which are known as *generic functions*, produce different types of output, depending on the class of input. We will use sans serif font to distinguish R classes from objects.

Data and definitions for specific polygons can be extracted by various methods and printed for inspection to facilitate error-checking. The following commands, for example, return the class of `MN.grid`, then the row number (2) and data associated with polygon 1378, and finally use the row number as a subscript to extract the polygon definition. Because the polygon definition is quite lengthy, we have used `str()` (short for "structure") to print a summary rather than the definition itself. The function `str()` is a very useful tool because it provides a simple means for learning about the structure and contents of any object.

```
> class(MN.grid@data)

[1] "data.frame"

> subset(MN.grid@data, poly == "1378")

  poly     area perimeter group    acres  centr.x centr.y
2 1378 40593388  25626.18     A 10030.63 259924.3 5278418

> str(MN.grid@polygons[[2]])

Formal class 'Polygons' [package "sp"] with 5 slots
  ..@ Polygons :List of 1
  .. ..$ :Formal class 'Polygon' [package "sp"] with 5 slots
  .. .. .. ..@ labpt  : num [1:2]  259924 5278418
  .. .. .. ..@ area   : num 40593389
  .. .. .. ..@ hole   : logi FALSE
  .. .. .. ..@ ringDir: int 1
  .. .. .. ..@ coords : num [1:32, 1:2] 256861 258472 260035 260185 260953 ...
```

```
  ..@ plotOrder: int 1
  ..@ labpt    : num [1:2]  259924 5278418
  ..@ ID       : chr "1378"
  ..@ area     : num 40593389
```

Subsequent steps will require an ordered list of polygon labels. We will first use the extraction operator (`$`) to generate a character vector of labels, then subscripts to print the first 5 elements for inspection.

```
> polynames = MN.grid$poly
> polynames[1:5]
```

```
[1] "1360" "1378" "1379" "1380" "1381"
```

`GetNList()` is a function for identifying neighbors of polygons. Results are output as an R list with 1 component for each polygon in the survey region. Each component is a vector of labels for neighboring polygons.

Executing `GetNList()` may take several minutes for large grids. This is an unfortunate side-effect of the need to accomodate somewhat irregular shapes and sizes of polygons that were surveyed in Minnesota. For sufficiently regular grids, distances between centroids can be used to identify neighborhoods much more quickly.

In addition to `MN.grid`, arguments to `GetNList()` include 3 variables within `MN.grid@data`.[1] These include the unique identifier for each polygon (e.g., `poly`) and coordinates of polygon centroids (e.g., `centr.x`, `centr.y`). The argument ***`range`*** specifies the maximum distance between centroids of potential neighbors and excludes polygons from consideration if they are far apart, which helps to speed up processing. `GetNList()` operates by computing distances between regularly spaced points on polygon boundaries and declaring cells to be neighbors if the minimum distance is less than some tolerance. The number of regularly spaced points selected for each polygon boundary and the tolerance are specified by `n.pnts` and `tol`, respectively.

```
> N.list = GetNList(spdf = MN.grid, IDvar = "poly",
```

[1] Recall that MN.grid is an object of class SpatialPolygonsDataFrame; the `@` extraction operator returns the second component, which is named `data`.

```
+     xvar = "centr.x", yvar = "centr.y", range = 11000,
+     include.target = F, n.pnts = 50, tol = 1000)
```

> **Quotation marks**
> Quotation marks distinguish character strings from R objects. In the preceding example, the argument `spdf` requires an R object of class SpatialPolygonsDataFrame (e.g., `MN.grid`), whereas IDvar requires a string of characters that names an object within `MN.grid`.

Checking the list of neighborhoods for errors is advisable, so we will compute the number of neighbors for each polygon in `MN.grid`. Polygons in the grid are roughly square, so we expect each to have from 3 (polygons on corners) to 8 (interior polygons) neighbors. The following commands first compute the number of neighbors for each polygon (given by the length of each component of `N.list`), then convert the output to a vector and print the first 5 values.

```
> N.n = lapply(N.list, length)
> N.n = unlist(N.n)
> N.n[1:5]
```

```
1360 1378 1379 1380 1381
   4    5    5    5    4
```

We can now use `table()` to compute frequencies of occurrence for various numbers of neighbors and show that numbers of neighbors were within the expected range. In this case, 9 polygons had 3 neighbors, 17 polygons had 4 neighbors, etc.

```
> table(N.n)
```

```
N.n
  3   4   5   6   7   8
  9  17 150  23  45 982
```

`PlotPolys()` facilitates the preparation of color-coded maps of survey grids and provides a convenient means for error-checking counts of neighbors. To demonstrate, we will use `PlotPolys()` to generate a map of the survey grid, with polygons color-coded to represent numbers of neighbors computed by `GetNList()` (Fig. 1). Arguments will include the study area description to be plotted (`MN.grid`), the name of a vector of unique polygon labels (`"poly"`) in `MN.grid`, a vector of labels for cells to be colored (`polynames`), and a vector of colors (`N.n`). R permits colors to be specified in several ways: in this case, we have used numeric indices that reference the default color palette.

`PlotPolys()` alone will produce a very nice plot; in addition, we will follow up by creating 2 temporary objects (a vectors of label, `lbls`, and a vector of colors, `clrs`) and using them to add a legend with `legend()`. We will then use `rm()` to remove the temporary objects after they have served their purposes.

```
> PlotPolys(spdf = MN.grid, IDvar = "poly", polys = polynames,
+     col = N.n)
> lbls = names(table(N.n))
> clrs = as.numeric(lbls)
> legend(x = 175000, y = 4872000, legend = lbls,
+     fill = clrs, cex = 0.65)
> rm(lbls, clrs)
```

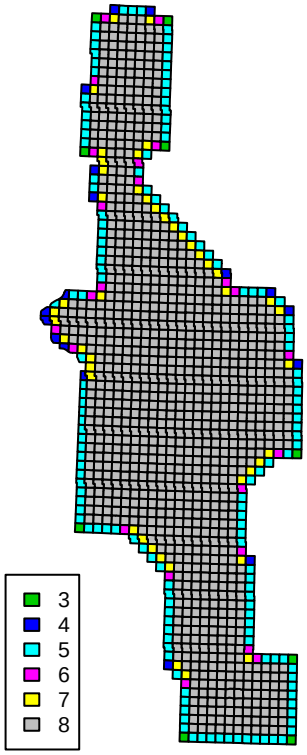

Figure 1: *Carnivore survey grid in western Minnesota. Colors represent numbers of neighbors for polygons in the survey region.*

`PlotN()` provides for more thorough inspections by plotting neighborhoods for individual polygons. Required arguments include ***spdf*** (a SpatialPolygonsDataFrame created with `ReadGridDef()`), ***IDvar*** (the unique identifier for polygons in ***spdf***), ***N*** (a list of neighbors for polygons in ***spdf***, created with `GetNList()`), and ***polys*** (character string or vector naming the polygon(s) to be included). Output can be routed to the screen (the default) or to a `.pdf` file (if an optional ***file*** argument is supplied). If more than 1 plot is requested and output is routed to the screen, the first plot will be displayed and the user will be prompted to press <*Enter*> to view succeeding plots or <*Esc*> to exit.

The command below produces a plot of the neighbors for polygon 3381, as shown in Fig. 2:

```
> PlotN(spdf = MN.grid, IDvar = "poly", polys = "3381",
+     N = N.list)
```

> The CD containing this report includes `Neighborhoods.pdf`, which contains a neighborhood plot for each of the 1226 polygons in the survey region.

Either `ls()` or `objects()` will generate a list of objects that are available for use within the current R session. At this point, our workspace includes 3 objects that will be required for subsequent analyses:

```
> ls()
```

```
[1] "MN.grid"   "N.list"    "N.n"       "polynames"
```

Table 2: *Summary of objects created to describe the survey region.*

Object	**Class**	**Description**
`MN.grid`	SpatialPolygonsDataFrame	A definition of the survey region.
`N.list`	List	A list with one component for each polygon in the survey region. Each component identifies neighbors of a polygon in the survey region.
`polynames`	Vector	Names for polygons in the survey region.
`N.n`	Vector	Numbers of neighbors for polygons in the survey region.

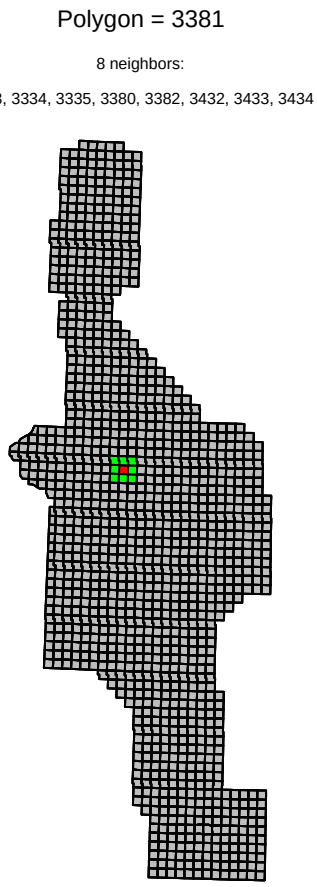

Figure 2: *Example of output produced by* `PlotN()` *to facilitate detailed inspections of neighborhoods.*

4 Importing and organizing survey data

Survey results ultimately will be represented by 1 record (a search and detection history) for each polygon in the survey region. Search histories and detection histories are typically represented by strings of binary variables: for example, a polygon with a search history of 110 was searched on the first and second of 3 search occasions. A corresponding detection history of 010 denotes an encounter of the target species on the second occasion, but not the first or the third.

Entering detection histories is inconvenient and errors are likely to result. To facilitate data entry, we constructed a function (`FormatData()`) for importing and reorganizing data stored in a more convenient format.

`FormatData()` requires a comma-separated text (`.csv`) file like the one shown below and described in Table 4. Key elements of the example include a header line of variable names; a variable that uniquely identifies each polygon in the survey region (`poly`); required variables (`occasion`, `mm`, `dd`, `yyyy`, and `start.time`), detection times for species of interest (e.g., red fox [*Vulpes vulpes*; `fox`] and coyote [*Canis latrans*; `coyote`]); and one auxiliary variable (observer; `obs`).

Table 3: *Contents of a sample* `.csv` *file containing survey data (commas not shown).*

	poly	occasion	mm	dd	yyyy	start.time	fox	coyote	observer
1	1360	1	6	21	2005	13:45:00	<NA>	14:38:00	TJT
2	1360	2	7	29	2005	14:15:00	14:21:00	14:19:00	TJT
3	1379	1	6	21	2005	10:20:00	<NA>	<NA>	TJT
4	1379	2	7	14	2005	14:30:00	<NA>	14:42:00	TJT
5	1381	1	6	13	2005	9:35:00	10:49:00	<NA>	TJT

The data file should be stored, along with the shapefile describing the survey region, in the working directory to be used for analyses.

`FormatData()` imports the data, converts dates and times to a standard format that supports mathematical operations (e.g., subtraction to find elapsed time), and generates records for polygons that were not surveyed. Required arguments include a character string naming a data source file (e.g., `Example data.csv`), the name of a variable that uniquely identifies each polygon (`IDvar`), and a vector of permissible polygon names (`all.polys`). Optional arguments include ***`exclude`***, a vector of character strings naming variables to be excluded when the data are imported, and ***`dur`***, the duration of search periods (fraction of a day).

Table 4: *Required* `.csv` *file structure for importation with* `FormatData()`

Records	
Variable names on the first row followed by 1 record per *search* of a sample cell	
Variables	*Description*
identifier (e.g., `poly`)	Unique identifier for each polygon in the survey region
`occasion`	Numeric index to search occasion, 1-J for J occasions
`mm`	Month number, 1-12
`dd`	Date, 1-31
`yyyy`	Four-digit year
`start.time`	Time search initiated; h:m; (00:00 - 23:59)
detection times (e.g., `fox`, `coyote`)	hh:mm if found, otherwise `NA`
Missing values	
Coded `NA` (**N**ot **A**vailable); allowed only for detection times, where they indicate species that were not detected, and for auxiliary variables	

If a value is specified for ***dur***, detections that did not occur within a scheduled search interval will be disallowed. This option can be useful if observers did not adhere strictly to the specified length of search periods when recording detections. Note that ***dur*** may be specified either within `FormatData()`, or during the next step, within `MakeHist()`.

For purposes of this example, we will first import a sample data file containing survey results for 2 species (fox and coyote), then print the first 5 records:

```
> example.data = FormatData(filename = "Example data.csv",
+     IDvar = "poly", all.polys = polynames, exclude = "observer",
+     dur = 1.25/24)
```

```
> example.data[1:5, ]
```

```
  poly occasion          start.time                fox                coyote
1 1360        1 (06/21/05 13:45:00)           (NA NA) (06/21/05 14:38:00)
2 1360        2 (07/29/05 14:15:00) (07/29/05 14:21:00) (07/29/05 14:19:00)
```

```
3 1378        1             (NA NA)             (NA NA)             (NA NA)
4 1378        2             (NA NA)             (NA NA)             (NA NA)
5 1379        1 (06/21/05 10:20:00)             (NA NA)             (NA NA)
```

`MakeHist()` facilitates the construction of search and detection histories from data like those we imported with `FormatData()`. Required arguments include a dataframe generated by `FormatData()` and the names of 2 constituent variables: *`IDvar`* uniquely identifies each polygon and *`detect.var`* contains times of detection for a target species (in this case, we will use `fox`).

```
> sample.hist = MakeHist(data = example.data, IDvar = "poly",
+     occ.var = "occasion", start.var = "start.time",
+     detect.var = "fox")
> sample.hist[1:5, ]
```

Records indicate, for example, that polygon 1360 was searched on 2 occasions (`s.1`=1, `s.2`=1, `n.s`=2) and foxes were found only on 1 occasion (the second; `d.1`=0, `d.2`=1, `n.d`=1).

`AddPolyData()` simplifies and error-checks the addition of variables to the dataframe component of a SpatialPolygonsDataFrame (e.g., `MN.grid`). We will use `AddPolyData()` to combine search and detection histories with the description of our survey region, so that all of the data associated with our analysis is stored in a single object, `sample.spdf`.

```
> sample.spdf = AddPolyData(spdf = MN.grid, add = sample.hist,
+     IDvar = "poly")
```

Note above that `AddPolyData()` issued a warning because `sample.hist` included polygon labels not found in `MN.grid`. Such a result was expected and appropriate in this case because several polygons were eliminated from the definition of the survey region after data were collected and entered. More generally, however, the warning may be an indication of data entry errors leading to losses of records.

To reduce the potential for confusion, we can now use `rm()` to remove objects that will not be required for subsequent analyses.

```
> rm(example.data, sample.hist, MN.grid, N.n)
```

Our workspace still includes only 3 objects: `MN.grid` has been replaced by `sample.spdf`, which includes both a description of the survey region and detection histories for species 1.

```
> ls()
```

```
[1] "N.list"      "polynames"   "sample.spdf"
```

Table 5: *Summary of objects containing survey data.*

Object	**Class**	**Description**
`sample.spdf`	SpatialPolygonsDataFrame	A definition of the survey region with a sample of search and detection histories attached.
`N.list`	list	A list with one component for each polygon in the survey region. Each component identifies neighbors of a polygon in the survey region.
`polynames`	character	A vector of names for polygons in the survey region.

5 Model fitting

5.1 Methods

Model fitting will be accomplished by Markov chain Monte Carlo (MCMC) simulation (Gilks et al. (1996)) with an algorithm described by Sargeant et al. (2005). Ultimately, the model fitting algorithm will estimate 1) a range map represented by a vector of numeric values with 1 element for each polygon in the survey region, 2) a smoothing coefficient, β, that controls the weight of evidence granted to neighbors when computing map values for polygons, and 3) θ, the per-search probability of detecting the species of interest when it is present.

Procedures demonstrated in this report differ in 1 key respect from those of Sargeant et al. (2005): for swift fox surveys conducted in Kansas, searches of sample polygons were suspended when swift foxes were detected. Numbers of searches were thus random variables. In Minnesota, each survey unit was searched twice. This difference in procedures necessitates a modification of the likelihood for each polygon. We thus replaced equation 2 of Sargeant et al. (2005) with a likelihood based on the binomial distribution:

$$\ell(d_i|x_i = 1, s_i, \theta) = \theta^{d_i}(1-\theta)^{s_i - d_i}$$

Notation represents the per-search probability of detection (θ) the unknown true status(x_i; occupied (1) or unoccupied (0)), the number of searches (s_i), and the number of detections (d_i) for polygon i. Underlying assumptions are the same as those discussed at length by Sargeant et al. (2005).

5.2 Model fitting

We will store the input required for model fitting in a list constructed with `InputList()`. Arguments to `InputList()` will include a dataframe (*`data`*), a list of neighborhoods constructed with `GetNList()`(*`N`*), a numeric vector of initial values for the map (*`map.0`*), initial values for model parameters (*`beta.0`* and *`theta.0`*), standard deviations for proposal distributions (*`sd.beta`* and *`sd.theta`*), and a seed (*`seed`*) for the random number generator in R (specifying the seed makes it possible to duplicate results generated by stochastic simulation). Contents of *`data`* must include three variables: `poly` (a unique identifier for each polygon), `n.s` (the number of searches for each polygon), and `n.d` (the number of detections

for each polygon).

Initial map values must be provided as a vector with **named** elements, and names **must** correspond with polygon names. Entries may be either 0 or 1 and can be chosen arbitrarily subject to this constraint: we will use a vector of zeroes constructed with `rep()`. Initial values for β and θ can be chosen arbitrarily within the interval (0,1). Standard deviations must be chosen by trial and error. Having run the analysis previously, we have the advantage of "inside information" and will choose values to facilitate a discussion of convergence and mixing (see 5.3).

```
> n.polys = length(polynames)
> map.in = rep(0, n.polys)
> names(map.in) = polynames
> input.1 = InputList(data = sample.spdf@data, N = N.list,
+     map.0 = map.in, beta.0 = 0.4, theta.0 = 0.5,
+     sd.beta = 0.25, sd.theta = 0.25, seed = 1)
> input.2 = InputList(data = sample.spdf@data, N = N.list,
+     map.0 = map.in, beta.0 = 0.4, theta.0 = 0.5,
+     sd.beta = 0.005, sd.theta = 0.005, seed = 1)
```

We will accomplish model fitting with a utility called `FitModel()`. Required arguments include a list created by `InputList()` and the desired number of iterations to be executed. We will initially conduct 2 very short runs (100 iterations). These will provide some assurance that our model-fitting algorithm is working properly before we invest the time required to generate the thousands of iterations required for estimation.

By default, `FitModel()` prints the iteration number, elapsed time, and an estimate of time to completion at end of each cycle. This information is reassuring and useful when analyses may take minutes or hours to complete but the volume of output can be troublesome when output is routed to a printed report. We have thus suppressed tracking information by supplying an optional argument (`tracker=F`).

```
> output.1 = FitModel(input.1, J = 100, tracker = F)
> output.2 = FitModel(input.2, J = 100, tracker = F)
```

Results of model fitting are output as a list that includes the call to `FitModel()` (`call`; documents the source file used for input); the time of execution (`time`; initiation and completion); either `map.means` or `maps` (see next paragraph); `beta` (a matrix containing proposed

and accepted values for β); and `theta` (a matrix containing proposed and accepted values for θ).

The nature of map output is determined by the number of iterations executed during model fitting. For $J < 100$, output includes `maps`, a list of vectors, each of which is a map generated during 1 iteration: each vector thus contains a 0 (species absent) or 1 (species present) for each polygon in the survey region. For $J \geq 100$, output includes `map.means`, a list with components that each record the average of 100 maps. Limiting output to summaries speeds processing considerably for large maps and large numbers of iterations.

5.3 Convergence and mixing

Model fitting proceeds by proposing a new state (occupied or unoccupied) for each polygon and new values for model parameters during each iteration. These proposals are either accepted or rejected according to certain rules of probability (see Sargeant et al. 2005). This procedure ultimately produces dependent series of maps (1 per iteration) and parameter estimates that each are consistent with the data and model, and current estimates of other quantities.

Initial values declared at the outset of model fitting, however, are not likely to be consistent with the data or model and should not be allowed to influence estimates. Sequences of maps and parameters may take many cycles to converge from initial values to the target distribution to be estimated.

`Convergence()` is a utility that facilitates the inspection of series of estimates for model parameters. It requires 2 arguments–a numeric vector containing a sequence of estimates and a label for the y axis–and is invoked as shown below. Sample output is shown in Fig. 3.

```
> Convergence(model.output = output.1)
```

Axis labels may be text (specified as a character string, e.g., `ylabel="text"`) or include special symbols (e.g., Fig. 3). Instructions for programming special symbols can be generated with `demo()`:

```
> demo(plotmath)
```

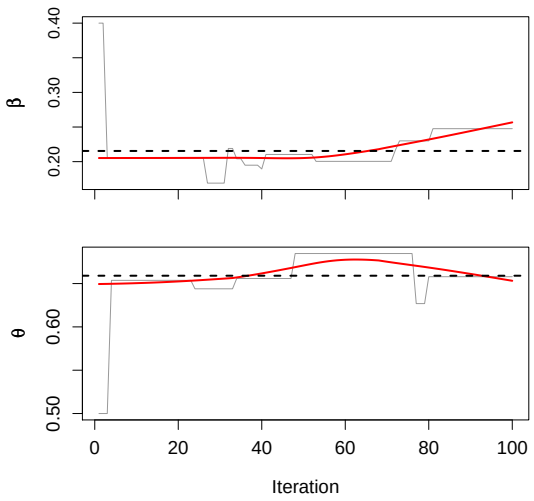

Figure 3: *Series of parameter estimates for θ and β from* `output.1`*. Dashed reference line denotes the mean. Red line depicts moving average resulting from a lowess smooth.*

Fig. 3 reflects a problem that we caused intentionally, for purposes of illustration, by choosing unsuitably large values for `sd.theta()` and `sd.beta()` during model fitting. Standard deviations control the spread of proposals around the current value. Very large values lead to many proposals that are rejected because they are much less credible than the current value. As a result, sequences of estimates become "stuck" and do not "mix," or move very rapidly throughout the posterior distribution we are attempting to estimate. Estimation is very inefficient in such cases and takes a prohibitively (and unnecessarily) large number of iterations.

Very small values can produce the same problem (slow mixing), but for different reasons. Proposals often are accepted but are invariably very close to current values, so estimates of model parameters change very slowly. Series of estimates thus exhibit very long, gradual trends (Fig. 4).

With these results in mind, we will construct a new list for input and output, specify intermediate values for standard deviations, and refit the model. We know from experimentation that standard deviations of 0.02 will lead to the acceptance of 60-70% of proposals, steady

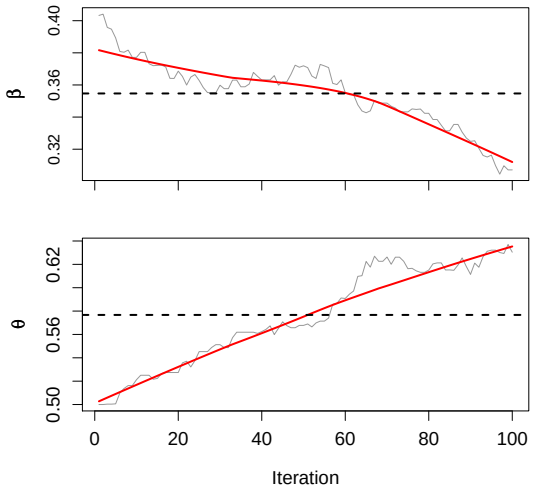

Figure 4: *Series of parameter estimates for θ and β from* `output.2`*. Dashed reference line denotes the mean. Red line depicts moving average resulting from a lowess smooth.*

convergence, and rapid mixing. We also will specify 25000 iterations so distributions of values generated by simulation will closely match the target posterior distributions.

```
> input.3 = InputList(data = sample.spdf@data, N = N.list,
+     map.0 = map.in, beta.0 = 0.4, theta.0 = 0.5,
+     sd.beta = 0.02, sd.theta = 0.02, seed = 1)
> output.3 = FitModel(input.3, J = 25000, tracker = F)
```

As expected, parameter estimates now converge rapidly and mix more quickly than in previous examples (Fig. 5). Note below that for the first time we have supplied an optional argument (`burn.in`), which excludes the first 5000 iterations when calculating the position of the dashed reference line.

```
> Convergence(model.output = output.3, burn.in = 5000)
```

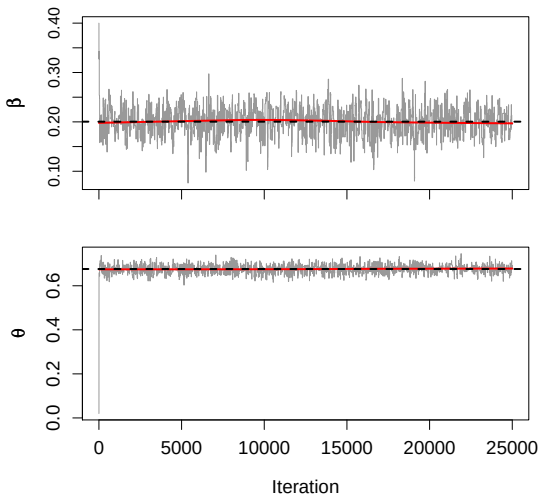

Figure 5: *Series of parameter estimates for θ and β from* ***output.3*** *. Dashed reference line denotes the mean (first 5000 iterations excluded). Red line depicts moving average resulting from a lowess smooth.*

5.4 Other diagnostics

5.4.1 Acceptance rates

Given generally positive results, a closer inspection of acceptance rates is in order. `Acceptance()` is a utility for computing acceptance rates for proposals. The only required argument is an object produced by `FitModel()`. As above, however, we will supply an optional argument (***burn.in***=100) to exclude the first 100 iterations from consideration when computing the acceptance rate. As expected, about 2/3 of proposals were accepted.

```
> Acceptance(output.3, burn.in = 5000)
```

```
      accepted rejected
theta   0.0829   0.9171
beta    0.0807   0.9193
```

5.4.2 Posterior distributions

For a sufficiently large grid, we expect sequences of parameter estimates to converge to approximately normal distributions. We have thus provided 2 utilities for inspecting values in sequences for θ and β. `Normality()` generates normal probability plots (Fig. 6. `Distribution()` plots densities along with corresponding normal densities (same mean and standard deviation; Fig. 7).

```
> Normality(output.3, burn.in = 5000)
```

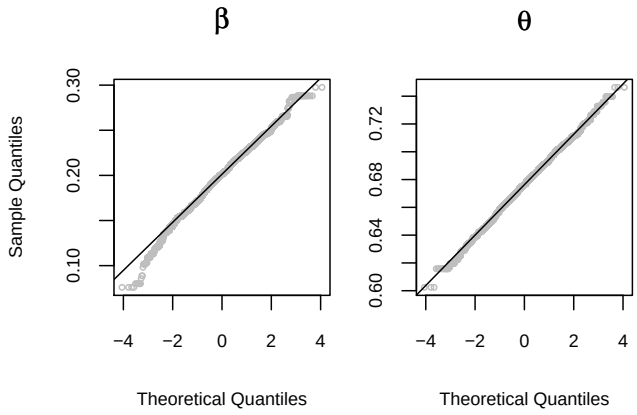

Figure 6: *Normal probability plots for series of parameter estimates from* `output.3`.

```
> Distribution(output.3, burn.in = 5000)
```

```
[1] "mean and se for beta 0.2002 0.0269"
[1] "mean and se for theta 0.6757 0.0188"
```

Finally, we can use `PlotSampleProbs()` to examine the convergence and distribution of estimated probabilities of occurrence for sample polygons selected at random (Fig. 8). An optional argument (`burn.in`) excludes the first 5000 iterations from consideration.

```
> PlotSampleProbs(output.3, burn.in = 5000)
```

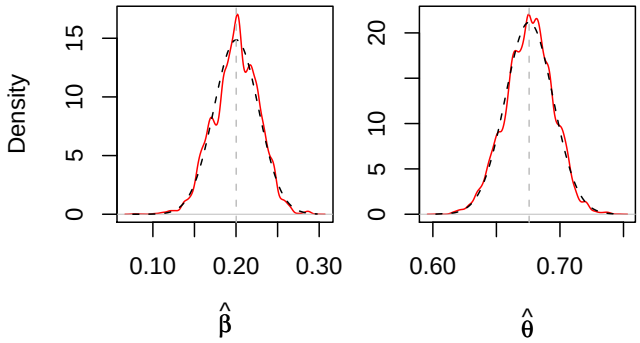

Figure 7: *Densities (in red) for series of parameter estimates from* `output.3`*. The dashed reference line depicts normal distribution with the same mean and standard deviation.*

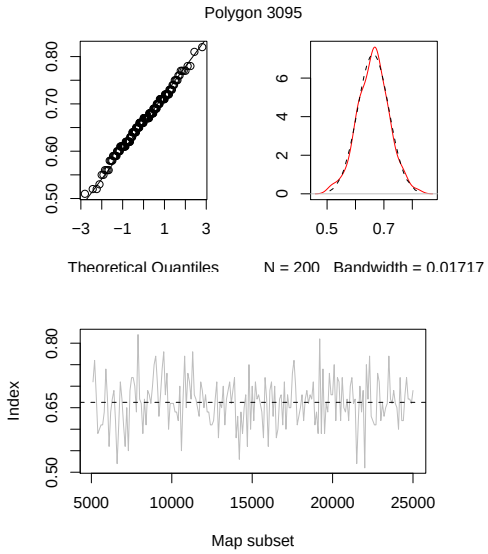

Figure 8: *Normal probability plot, density plot, and series of estimated probabilities of occurrence for a randomly selected map polygon from* `output.3`*.*

6 Results

6.1 Estimates of β and θ

Markov chain Monte Carlo simulation is a strategy for drawing a long sequence of dependent random variables (a Markov chain) from some distribution of interest. Properties of the sequence are then used to estimate properties of the target distribution.

In addition to functions that generate graphical summaries (previous section), we have provided a function called `ChainStats()` for estimating properties of β and θ from chains included in model output. We have included means, standard errors and a selection of quantiles. Quantiles estimate familiar summary statistics, including the range (0th and 100th), 99% Bayesian confidence limits (0.5th and 99.5th), 95% Bayesian confidence limits (2.5th and 97.5th), the interquartile range (25th and 75th), and the median (50th). In addition, we have included the 33rd and 67th: for normally distributed data, we expect these to correspond with the mean $\pm$ 1 standard error. Note the use of an optional argument (burn.in) to exclude values generated prior to convergence.

```
> ChainStats(output.3, burn.in = 5000)

             beta      theta
mean   0.20041665 0.67606357
se     0.02742755 0.01844209
0%     0.07594637 0.60241840
0.5%   0.12207205 0.62758900
2.5%   0.14640606 0.63938674
25%    0.18331231 0.66406287
33%    0.19007872 0.66820411
50%    0.20116175 0.67639037
67%    0.21378616 0.68424035
75%    0.21923080 0.68852919
97.5% 0.24912439 0.71087747
100%   0.29753503 0.74635604
```

Standard deviations and standard errors are often misunderstood. Recall that standard errors are standard deviations for sampling distributions of statistics. In this case, standard errors are estimated by standard deviations of chains produced during model fitting.

6.2 Indices of occurrence

For each iteration, `FitModel()` generates a map of zeroes (indicating absence) and ones (indicating presence). Model output includes either these original maps (when $J < 100$) or interim averages for sets of 100 maps (when $J \geq 100$). To index probabilities of occurrence, we will use proportions of occurrences (ones) across all of the maps generated during model fitting (excluding those generated prior to model convergence).

We will use `GetIndex()` to extract index values from model output, then `AddPolyData()` to combine the result with our study area description and input data.

```
> sample.index = GetIndex(output.3, burn.in = 5000)
> results.spdf = AddPolyData(spdf = sample.spdf,
+     add = sample.index, IDvar = "poly")
```

Finally, we can prepare maps in R and use `writePolyShape()` to export results to a shapefile (`Sample.output`) for use with other software systems. `DrawMap()` simplifies the preparation of maps and includes options for controlling such features as the color scheme (*color* = `"topo"`, `"heat"`, or `"gray"`), the scale used for plotting (*scale* = `"data"` or `"percentile"`), and the size of the legend (*leg.width*; a numeric expansion factor).

Components of our exported shapefile will appear in our working directory and will include `Sample output.dbf`, `Sample output.shp`, and `Sample output.shx`. Indices of occurrence also appear in Fig. 9.

```
> DrawMap(spdf = results.spdf, IDvar = "poly", est = results.spdf@data$index,
+     color = "topo", scale = "data", legend = T,
+     leg.width = 1)
```

```
> writePolyShape(results.spdf, "Sample output")
```

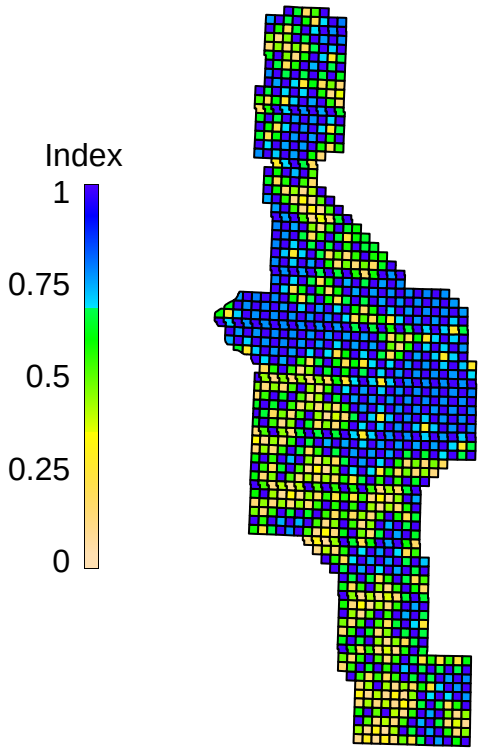

Figure 9: *Sample rangemap generated with* `DrawMap()` *from the study area description and index values included in* `results.spdf`.

References

P. Dalgaard. *Introductory statistics with R.* Springer, New York, New York, USA, 2002.

W. R. Gilks, S. Richardson, and D. J. Spiegelhalter. Introducing Markov chain Monte Carlo. In W. R. Gilks, S. Richardson, and D. J. Spiegelhalter, editors, *Markov chain Monte Carlo in practice*, pages 1–19. Chapman and Hall/CRC, Boca Raton, Florida, USA, 1996.

J. Heikkinen and H. Högmander. Fully Bayesian approach to image restoration with an application in biogeography. *Journal of the Royal Statistical Society, series C*, 43:569–582, 1994.

G. A. Sargeant, M. A. Sovada, C. C. Slivinski, and D. H. Johnson. Markov chain Monte Carlo estimation of species distributions: a case study of the swift fox in western Kansas. *Journal of Wildlife Management*, 69:483–497, 2005.

M. A. Sovada and G. A. Sargeant. Monitoring mammalian predator distributions and development of techniques to assess abundance of predator species. Unpublished report, 2007.

R Development Core Team. *R: a language and environment for statistical computing.* Foundation for Statistical Computing, Vienna, Austria, 2003.

Appendix: Acquiring and installing R

These instructions describe the installation of R on a Windows (95 or later) operating system.

1. Use Windows Explorer to create a subdirectory called `c:\R`.
2. Use your web browser to visit the "R Project For Statistical Computing" at `http://www.r-project.org/` and download a copy of the current version of R.
   (a) Under "Getting started", select the link to choose a CRAN Mirror. Your internet browser will be redirected to a list of mirror sites.
   (b) Scroll down the list of mirror sites to find sites in the United States. Select a site near you (e.g., `http://cran.cnr.Berkeley.edu`). Your web browser will be redirected to a page titled "The Comprehensive R Archive Network."
   (c) Select an operating system from the list of precompiled binary distributions [e.g., Windows (95 and later)]. Your web browser will be redirected to a page titled, e.g., "R for Windows."
   (d) Select base to download binaries for the base distribution of R. Your web browser will be redirected to a page titled, e.g., "R 2.5-0 For Windows."
   (e) Select the link for the current version of R (e.g., rw2011.exe). A popup will appear, asking whether you want to run, save, or cancel the download. Select save.
   (f) A dialogue box will appear and you will be asked to specify a location for storing the download: use `c:\R`. The file is rather large (28.5 MB for R v. 2.5.0), so the download may take 10-20 minutes.
3. Close your web browser and execute the file downloaded in step 2f (e.g., "`R-2.5.0-win32.exe`").
   (a) In Windows, this can be accomplished either by selecting "Run" under the start menu and entering the name of the downloaded file or by selecting the program icon in Windows Explorer.
   (b) Follow the installation program instructions by approving installation, accepting the license agreement, and approving default settings *except* you should direct R to install in `C:\R\` and not in `c:\Program files\R\`. Some programs that augment R may not work properly if R is stored in a subdirectory with a name that includes spaces.

4. Major upgrades to R are typically released twice annually, during spring and fall. Patches (bug fixes) are released as needed.

Index