Using `edgefx` to compute the effects of habitat edges on a landscape

Emma E. Goldberg & Leslie Ries

March 8, 2010

Contents

1 Background

The edges of habitat patches affect species that live within the habitat. Often, edge effects are modeled as a simple function of distance to the nearest edge. Edge structure can be much more complex than that single-valued characterization, however. This package allows computation of edge effects due to all (or just nearby) edges in a landscape. The models implemented in this package are an extension of a model proposed by J. Malcolm (1994; Ecology 75:2438-45).

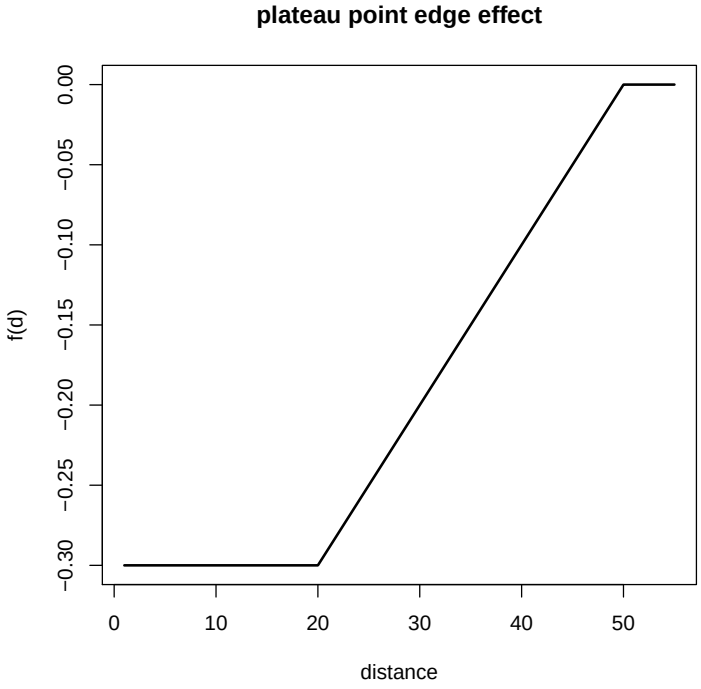
```
> library(edgefx)
```

Consider a response variable describing something about a species across space. For example, this could be the density of individuals, or the height of plants. Call it z. Say there is a baseline value far from any edges, k, and that edge effects cause deviations (either positive or negative) from this. Consider a particular point on the landscape where z is or could be measured; call it the "focal" point. Consider a second point that lies along a habitat edge and is a distance d from the focal point; call it the edge point. The effect of the edge point on the focal point can be modeled simply as a "plateau point edge effect,"

$$f(d) = \begin{cases} e_0 & d \le D_0 \\ e_0 \left(1 + \frac{D_0 - d}{D_{max} - D_0}\right) & D_0 < d \le D_{max} \\ 0 & d > D_{max} \end{cases} \quad (1)$$

where e_0 characterizes the maximum effect, D_0 is the distance out to which the maximum effect is felt, and D_{max} is the maximum distance at which any effect is felt. To see what $f(d)$ looks like, use `point.edge.effect()`:

```
> params <- list(e0 = -0.3, Dmax = 50, D0 = 20)
> d <- seq(params$Dmax * 1.1)
> plot(d, sapply(d, point.edge.effect, params), type = "l", lwd = 2,
+     xlab = "distance", ylab = "f(d)", main = "plateau point edge effect")
```



The simplest way of dealing with edges is to consider only the distance to the nearest edge, d_{min}, for each focal point [at position (x, y)], yielding

$$z(x, y) = k + f(d_{min}). \tag{2}$$

But this ignores the effects of other edge points, many of which may also be nearby. More complete would be to sum over all edge points within distance D_{max}, Γ, yielding

$$z(x, y) = k + \int_{\Gamma} f(s)\, ds. \tag{3}$$

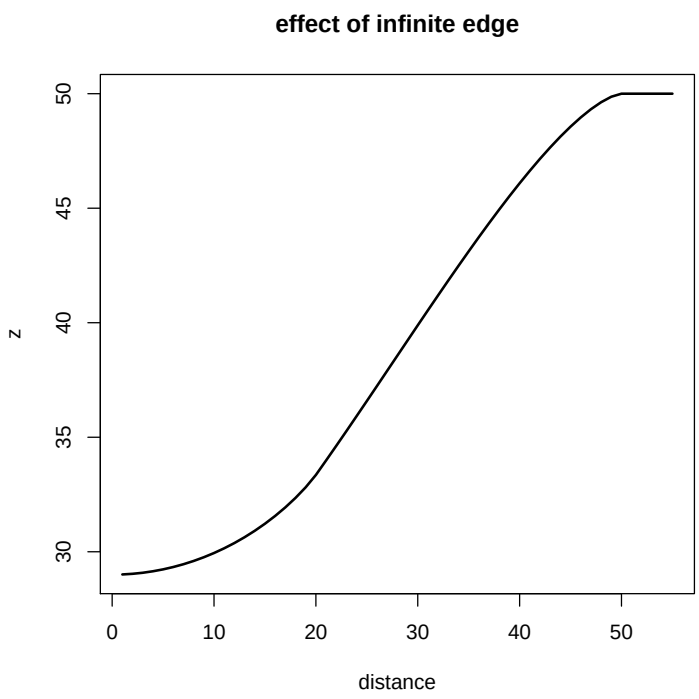
There are functions in `edgefx` to evaluate Eq. 3 for idealized infinite-extent edges, vector-based descriptions of finite habitat edges, and gridded habitat maps. Many of these procedures also allow estimation of the parameters in Eq. 1 from such data.

2 Infinite edges

2.1 Response prediction from given parameter values

As an idealized case, consider an edge that is linear and infinite in extent. To evaluate Eq. 3 at a range of distances from the edge, using the plateau point edge effect shown above, use `infinite.edge.effect()`:

```
> params$k = 50
> plot(d, sapply(d, infinite.edge.effect, params), type = "l",
+     lwd = 2, xlab = "distance", ylab = "z", main = "effect of infinite edge")
```



2.2 Parameter estimation from data

Suppose you have edges in your landscape that approximate linear, infinite edges (or they don't really, but you want to make that assumption for comparison purposes). If you have observed values of your response variable, z, at a variety of distances from infinite edges, d, you can fit for the parameters in the point edge effect function (Eq. 1).

To see this in action, first generate some fake data:

```
> params <- list(e0 = -0.3, Dmax = 100, D0 = 50, k = 50)
> d <- seq(0, 200, 1)
> set.seed(3)
> z <- sapply(d, infinite.edge.effect, params) + rnorm(length(d))
```

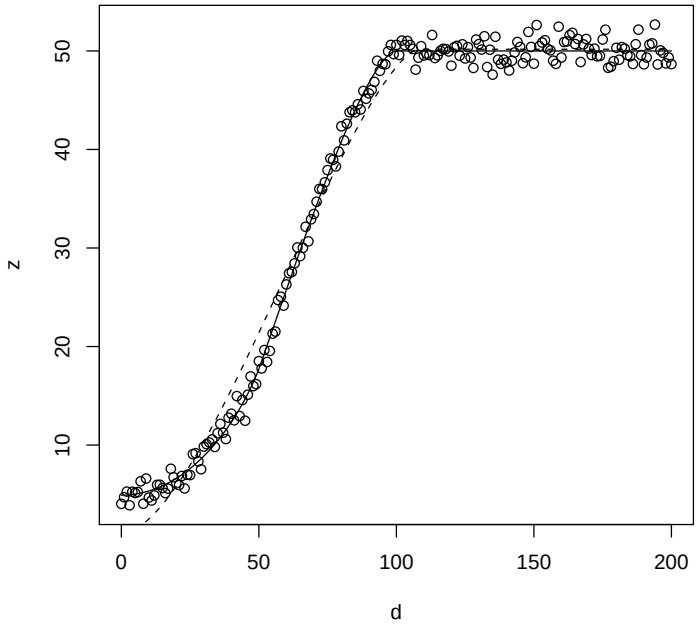
Now use non-linear least squares to fit the infinite edge function, with and without the D_0 parameter. You must provide some initial parameter guesses. Model convergence may be tricky, so best to run with several sets of initial values. Also, you may want to set the lower bound on `k` to 0 if that is what's physically appropriate for your response variable.

```
> nls.4par <- nls(z ~ sapply(d, infinite.edge.effect, e0, Dmax,
+     k, D0), start = list(e0 = -0.5, Dmax = 100, D0 = 40, k = 70),
+     algorithm = "port", lower = list(e0 = -Inf, Dmax = 0, k = -Inf,
+         D0 = 0))
> nls.3par <- nls(z ~ sapply(d, infinite.edge.effect, e0, Dmax,
+     k), start = list(e0 = -0.5, Dmax = 100, k = 70), algorithm = "port",
```

```
+     lower = list(e0 = -Inf, Dmax = 0, k = -Inf))
> summary(nls.4par)

Formula: z ~ sapply(d, infinite.edge.effect, e0, Dmax, k, D0)

Parameters:
      Estimate Std. Error t value Pr(>|t|)
e0   -0.301021   0.002279 -132.11   <2e-16 ***
Dmax 99.484352   0.560776  177.41   <2e-16 ***
D0   50.274987   0.906140   55.48   <2e-16 ***
k    49.997262   0.097512  512.73   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.9888 on 197 degrees of freedom

Algorithm "port", convergence message: relative convergence (4)

> summary(nls.3par)

Formula: z ~ sapply(d, infinite.edge.effect, e0, Dmax, k)

Parameters:
       Estimate Std. Error t value Pr(>|t|)
e0    -0.455647   0.005678  -80.25   <2e-16 ***
Dmax 107.938237   0.890337  121.23   <2e-16 ***
k     50.171318   0.197539  253.98   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.936 on 198 degrees of freedom

Algorithm "port", convergence message: relative convergence (4)
```

To see the fits:

```
> plot(d, z)
> lines(d, predict(nls.4par))
> lines(d, predict(nls.3par), lty = 2)
```



To perform an AIC test of the two models:

```
> AIC(nls.3par) - AIC(nls.4par)

[1] 269.0385
```

The four-parameter model fits better (lower AIC score), but not substantially so (difference is less than 2). (But be a bit careful about potential bugginess in the `nls` methods of `AIC` and `logLik`.)

See Section 3.3.1 for notes on the `nls` options and convergence messages. It also shows a different method of fitting infinite edges using fake maps.

3 Vectorized landscapes

A real habitat won't consist solely of an infinite linear edge, but it can be approximated as a collection of finite edge line segments. Once you've turned the habitat edges in your landscape into line segments, you can use those edge descriptions to (1) predict values of the response variable $z(x, y)$ using given parameter values (obtained through estimation or elsehow), and/or (2) use observed values of z to estimate the parameters in the point edge effect function, e_0, k, D_0, and D_{max}.

3.1 Input files

You should have a single input file for each focal point, containing the coordinates of the focal point and all the relevant edge segments. This package does not provide facilities for automatic identification of edges from general maps. Leslie will have notes on how best to get appropriate input files from GIS. But the per-focal-point files allow better customization of exactly which edge segments are "visible" to each focal point (e.g. don't include edges that are hidden behind other edges).

Here is an example of an input file:

```
# a comment
30, 50, ,    ,    focalhabitat
0,  0,  0,   100, edge1habitat
0,  0,  40,  20,  edge2habitat
40, 20, 100, 100, edge3habitat
```

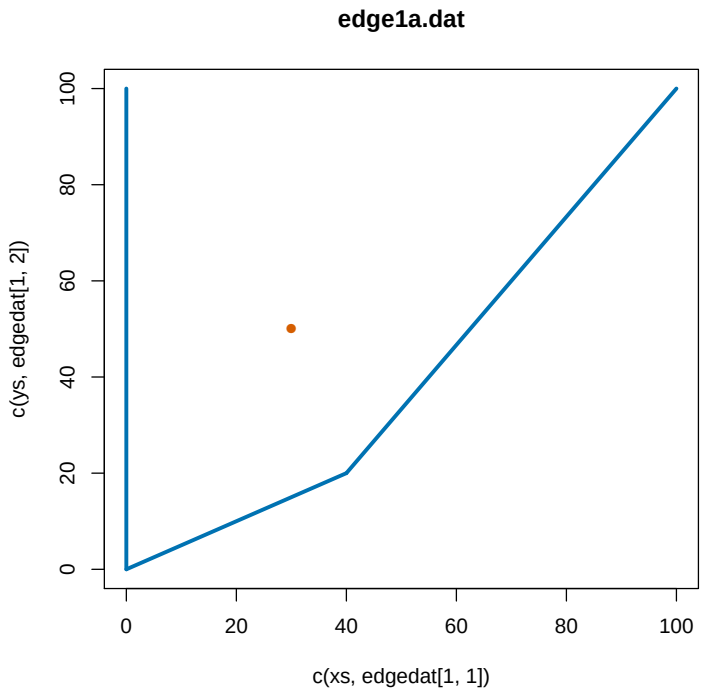
The origin of the coordinate system is arbitrary. The first non-comment line gives the x and y coordinates of the focal point; it needs two extra commas so R doesn't freak out about mis-matched numbers of columns. The rest of the lines are for the endpoints of each edge segment. The first edge extends from (0, 0) to (0, 100), the second from (0, 0) to (0, 40), etc. The last column is for notes about the habitat types, or whatever you want. Actually, you can have as many trailing columns as you want; their contents are ignored so far.

We've included a collection of simple edge input files for use here. You will have to identify where on your system they were installed; try looking somewhere like `/usr/local/lib/R/site-library/edgefx/doc/inputfiles/`; for the compilation of this documentation, it is just `inputfiles/`. You should define `prefix` to be wherever you find them. Let's read the example files into a list.

```
> prefix <- "inputfiles/"
> filenames <- c("edge1a.dat", "edge1b.dat", "edge1c.dat", "edge1d.dat",
+     "edge1e.dat", "edge2a.dat", "edge2b.dat", "edge2c.dat", "edge2d.dat",
+     "edge2e.dat")
> edgefilenames <- paste(prefix, filenames, sep = "")
> edgelist <- lapply(edgefilenames, read.table, sep = ",")
> names(edgelist) <- filenames
```

Our `edgelist` now has ten elements, one for each file. Here's what the first one looks like:

```
> edgelist[[1]]

  V1 V2  V3  V4           V5
1 30 50  NA  NA     habitat1
2  0  0   0 100     habitat2
3  0  0  40  20     habitat2
4 40 20 100 100     habitat2

> draw.edges(edgelist[[1]])
> title(names(edgelist)[[1]])
```



The focal point is shown in orange, and its edges are shown in blue.

In our example, the first five input files all have the same edges but different focal points, and the same for the second five. You can combine analysis for whatever files you want, provided that you expect (or are willing to assume) that the same parameter values apply to all of them.

3.2 Response prediction from given parameter values

Suppose that you have in hand a set of parameter values (possibly obtained from data fitting; see Section 3.3) and a vectorized map like the one just shown. To predict the value of the response variable at the map's focal point, given the edges in the map and the parameter values, use `vecmap.edge.effect()`:

```
> params <- list(e0 = -0.3, Dmax = 40, k = 100, D0 = 15)
> vecmap.edge.effect(edgelist[[1]], params)

[1] 88.80376
```

3.3 Parameter estimation from data

Now suppose that at each of the focal points, we have data on the value of the response variable z. Read in those observed values, along with the map names indicating which observations go with which set of edges.

```
> z.obs <- read.table(paste(prefix, "edgez.dat", sep = ""), header = T)
> z.obs

      map        z
1  edge1a 83.95431
2  edge1c 72.98649
3  edge1e 82.36007
4  edge2b 92.20720
```

```
5  edge2d 87.98545
6  edge1b 69.62626
7  edge1d 74.63733
8  edge2a 78.00661
9  edge2c 84.14898
10 edge2e 86.71185
```

Described next are two possible approaches for obtaining parameter estimates and their uncertainties from these data.

3.3.1 Non-linear least squares, via `nls()`

To estimate the parameter values, assuming normally-distributed errors, we can do a nonlinear least squares fit to the plateau point edge function integrated over all the edges for each focal point. So our dependent variable is `z.obs$z` and our independent variables come from the application of Eq. 3 to each matrix in `edgelist`. We need to provide a rough guess of the parameter values in order for `nls()` to get started. Again, the procedure is sensitive to starting parameters, especially when data are variable. We suggest using a range of starting values to determine if your models converge on the same solution.

```
> guess <- list(e0 = -0.1, Dmax = 80, k = 50, D0 = 5)
> edgefit <- edge.nls(edgelist, z.obs$z, guess)
```

You can pass additional arguments to `nls()` after `guess`, e.g., `trace=T`. To see how the fit did, use the result as you would any `nls` object, e.g.,

```
> summary(edgefit)

Formula: observed ~ by(edges[, 2:4], xvals, map.edge.effect, e0, Dmax,
    k, D0)[unique(xvals)]

Parameters:
     Estimate Std. Error t value Pr(>|t|)
e0   -0.27611    0.02547 -10.839 3.65e-05 ***
Dmax 42.47611    3.34888  12.684 1.47e-05 ***
k    97.20524    1.67003  58.206 1.73e-09 ***
D0   16.64407    3.40070   4.894  0.00273 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.692 on 6 degrees of freedom

Algorithm "port", convergence message: relative convergence (4)

> library(MASS)

> edgefit.profile <- profile(edgefit)

> confint(edgefit.profile)

           2.5%       97.5%
e0   -0.4700749  -0.2862819
Dmax 38.4377328  51.4741225
k    92.4350262 101.3855911
```

Unfortunately, the `port` algorithm is required in order to constrain D_0 and D_{max} to be non-negative (this is applied within `edge.nls`), but `port` is "unfinished" (according to the `nls` help page). One consequence is that the `profile` and `confint` functions don't work reliably with `nls` results when `port` is used. But you'll have the standard errors from `summary`, even if `confint` gives `NA`s.

Support for a non-zero value of D_0 is pretty strong in this example, but often it isn't. To fit Malcolm's original point edge effect function rather than the plateau function, just omit D_0:

```
> guess <- list(e0 = -0.1, Dmax = 80, k = 50)
> edgefit <- edge.nls(edgelist, z.obs$z, guess)

> summary(edgefit)

Formula: observed ~ by(edges[, 2:4], xvals, map.edge.effect, e0, Dmax,
    k)[unique(xvals)]

Parameters:
     Estimate Std. Error t value Pr(>|t|)
e0    -0.3746     0.0415  -9.027 4.18e-05 ***
Dmax  44.0714     3.4591  12.741 4.25e-06 ***
k     96.5783     1.9806  48.763 3.99e-10 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.942 on 7 degrees of freedom

Algorithm "port", convergence message: relative convergence (4)

> edgefit.profile <- profile(edgefit)

> confint(edgefit.profile)

           2.5%       97.5%
e0   -0.4700749  -0.2862819
Dmax 38.4377328  51.4741225
k    92.4350262 101.3855911
```

If you get a convergence message of anything other than `(0)` from `nls` (like the `(4)` we got here), it's best to try a variety of values for `guess`.

The function `edge.nls()` just provides a convenient wrapper to `nls()`. In case you want to tweak `nls`'s options yourself (or maybe try something else, like `nls2`), here are the extra steps to take. You can omit `D0` from `formula` and `guess` if you want. (Note: I don't know why `xvals` and `z` have to be defined separately, but R has a fit if they aren't.)

```
> edges <- relocate.edge.df(edgelist)
> xvals <- edges[, 1]
> z <- z.obs$z
> edge.formula <- formula(z ~ by(edges[, 2:4], xvals, map.edge.effect,
+     e0, Dmax, k, D0))
> guess <- list(e0 = -0.1, Dmax = 80, k = 50, D0 = 5)
> fit <- nls(edge.formula, start = guess, algorithm = "port", lower = list(e0 = -Inf,
+     Dmax = 0, k = -Inf, D0 = 0))
```

An alternative method for fitting to the infinite edge function (see Section 2.2) is to use the above procedure (or the one below, with `optim` or MCMC) but to replace the data frame of map information, `edges` above, with values that represent infinite edges, `inf.edges` here:

```
> n <- length(d)
> inf.edges <- data.frame(mapnames = paste("map", seq(n), sep = ""),
+     x0 = d, y1 = rep(-Inf, n), y2 = rep(Inf, n))

> head(inf.edges)

  mapnames x0   y1  y2
1     map1  0 -Inf Inf
2     map2  1 -Inf Inf
3     map3  2 -Inf Inf
4     map4  3 -Inf Inf
5     map5  4 -Inf Inf
6     map6  5 -Inf Inf
```

3.3.2 Generalized likelihood optimization and MCMC

An alternative to constrained optimization with `nls` is to deal with the (log)likelihood of the data directly, which the function `edge.lnL()` provides. For Gaussian errors, the log-likelihood is proportional to the sum-of-squared-differences, so using a general-purpose optimizer like `optim()` with `edge.lnL()` is in principle the same as using `nls()`, though you can specify different algorithms and ways to constrain parameter values. For Poisson errors, the likelihood function is computed slightly differently, but it can be used in the same way.

To use `optim()` directly (rather than through a wrapper as for `edge.nls()`) to obtain maximum likelihood parameter estimates, we have to abide by its rules. Specifically, the initial guess must be a vector (which can be obtained from a list by `unlist()`; if instead its elements are unnamed, they must be in the order e_0, D_{max}, k, and optionally D_0) and the value is minimized so the negative log-likelihood must be used (obtained from `edge.lnL()` with `neg=TRUE`). Additionally, it's a bit faster to pass `optim` the relocated edge dataframe (obtained via `relocate.edge.df()`) rather than the raw edge list.

Here is a sequence of examples:

```
> guess <- unlist(guess)
> edges <- relocate.edge.df(edgelist)
> optim(guess, edge.lnL, NULL, edges, z.obs$z, neg = T)

$par
          e0         Dmax            k           D0
  -0.3554509 1171.5757913  170.1661713   53.6657135

$value
[1] 367.7956

$counts
function gradient
     501       NA

$convergence
[1] 1

$message
NULL
```

Consult the `optim()` documentation to see that a convergence value of 1 indicates that the iteration limit has been reached. We can tell `optim()` to try for longer, though it turns out it was pretty close already:

```
> optim(guess, edge.lnL, NULL, edges, z.obs$z, neg = T, control = list(maxit = 1000))

$par
          e0         Dmax            k           D0
  -0.3553188 1171.2919167  170.1306333   53.6656399

$value
[1] 367.7954

$counts
function gradient
     507       NA

$convergence
[1] 0

$message
NULL
```

The convergence value of 0 now indicates that the optimization was successful. The maximum-likelihood parameter estimates are given by `$par`, but these values are quite different from those from the `nls()` fit. We can find a higher likelihood (lower `$value`) by using different starting values:

```
> guess <- c(e0 = -0.3, Dmax = 50, k = 100, D0 = 5)
> optim(guess, edge.lnL, NULL, edges, z.obs$z, neg = T)

$par
        e0       Dmax          k         D0
-0.2761111 42.4766116 97.2051586 16.6432461

$value
[1] 17.17024

$counts
function gradient
     237       NA

$convergence
[1] 0

$message
NULL
```

Now the agreement with the `nls` result is perfect.

When an optimization method is not specified, the default is Nelder-Mead and the parameter constraints are taken care of by returning a likelihood value of `-Inf` when $D_{max} < 0$ or $D_0 < 0$. You could instead use the quasi-Newton method with box constraints on the parameter values (see the `optim()` documentation, and be sure that the order in `lower` matches that in `guess`):

```
> optim(guess, edge.lnL, NULL, method = "L-BFGS-B", lower = c(-Inf,
+     0, -Inf, 0), edges, z.obs$z, neg = T)

$par
        e0       Dmax          k         D0
```

```
-0.2761129 42.4761864 97.2052726 16.6440494

$value
[1] 17.17024

$counts
function gradient
      71       71

$convergence
[1] 0

$message
[1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

To fix $D_0 = 0$, simply omit it from the guess:

```
> guess <- c(e0 = -0.3, Dmax = 50, k = 100)
> optim(guess, edge.lnL, NULL, edges, z.obs$z, neg = T)

$par
        e0       Dmax          k
-0.3746541 44.0709961 96.5786155

$value
[1] 26.39620

$counts
function gradient
     140       NA

$convergence
[1] 0

$message
NULL
```

Everything above also applies to Poisson errors, which you can request with `family="poisson"`. Note that your observed values must be positive integers—this should be intrinsically true for real data, but it must be forced in this illustration:

```
> guess <- c(e0 = -0.3, Dmax = 50, k = 100, D0 = 5)
> optim(guess, edge.lnL, NULL, edges, as.integer(z.obs$z), neg = T,
+     family = "poisson")

$par
        e0       Dmax          k         D0
-0.2801185 42.5425990 96.6676219 16.0719042

$value
[1] 31.28194

$counts
```

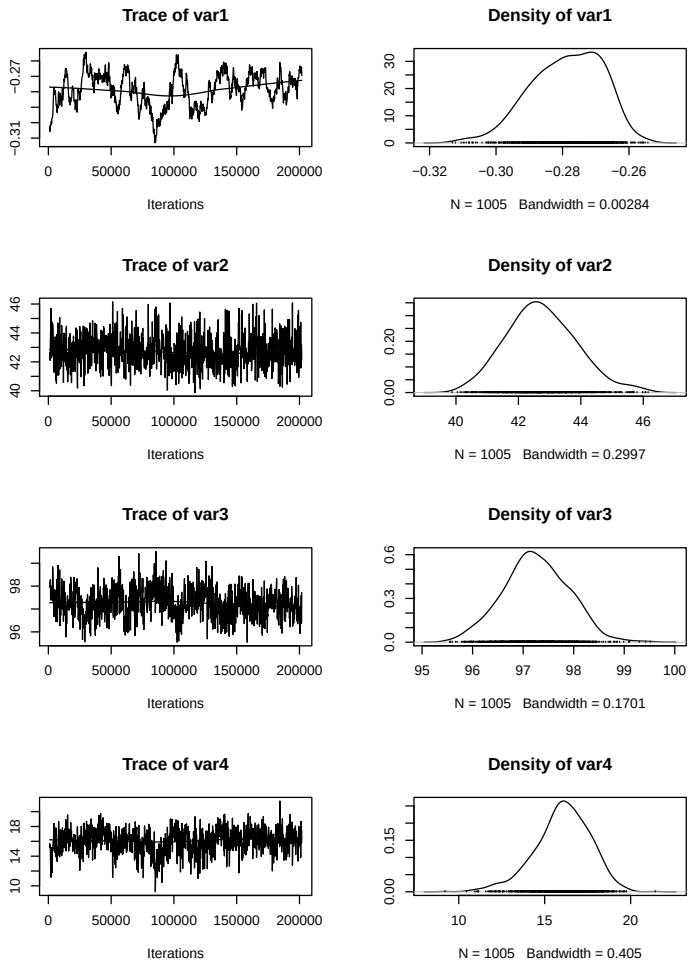
```
function gradient
     195       NA

$convergence
[1] 0

$message
NULL
```

Poisson errors often don't work well with `method="L-BFGS-B"` because negative predicted values must return a likelihood of `-Inf`, which this method apparantly can't handle.

Unlike `nls()`, `optim()` does not produce a structure for use by functions like `confint`. You could still map out the likelihood surface to get confidence intervals, but I couldn't find general-purpose R functions for that. Or you could try fitting with `optim()` and then use those as starting values in `nls()` to get results back as a nice structure.

Alternatively, issues like uncertainty and correlation in the parameter estimates can be addressed by obtaining posterior distributions from Markov chain Monte Carlo. You will have to learn about proper use of MCMC elsewhere, including diagnosing convergence, but here is an example to start with. (It could also take `family="poisson"`, presumably.) The initial values are informed by the `optim()` results, to reduce the burn-in time. The tuning values were chosen by experimentation to yield an acceptance probability of about 20%. The thinning interval was chosen after looking at autocorrelation plots (`acf()` is useful for this). It takes awhile to run.

```
> library(MCMCpack)

> guess <- c(e0 = -0.3, Dmax = 50, k = 100, D0 = 20)
> edgemcmc <- MCMCmetrop1R(edge.lnL, guess, burnin = 1000, mcmc = 201000,
+     thin = 200, tune = c(0.1, 1, 1, 1), optim.method = "Nelder-Mead",
+     edgecoord = edges, observed = z.obs$z)

> plot(edgemcmc)
```




```
> summary(edgemcmc)

Iterations = 1001:201801
Thinning interval = 200
Number of chains = 1
Sample size per chain = 1005

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

        Mean      SD  Naive SE Time-series SE
[1,] -0.2788 0.01068 0.0003368       0.001690
[2,] 42.7434 1.12665 0.0355390       0.042792
[3,] 97.2555 0.63961 0.0201759       0.042251
[4,] 16.0457 1.60456 0.0506142       0.114726

2. Quantiles for each variable:

        2.5%     25%     50%     75%   97.5%
var1 -0.3015 -0.2861 -0.2781 -0.2704 -0.2618
```

```
var2 40.6454 41.9654 42.6749 43.4879 45.2359
var3 95.9890 96.8341 97.2294 97.6966 98.4394
var4 12.3598 15.1368 16.1176 17.1772 18.8182
```

4 Gridded landscapes

As mentioned above, `edgefx` does not have sophisticated edge-detection functions. For a gridded landscape, the most it will do is identify as edges the cells of non-habitat ("matrix," but I'll avoid that term since it is also an R data structure) that have, among their four neighboring cells, at least one habitat cell. The response value z at each cell in the landscape can then be predicted as the sum of effects from all identified edge cells.

The distance between two cells is defined to be 1 for adjacent cells, and can be found for any pair of cell coordinates like so:

```
> distance(c(2, 3), c(4, 7))
```

```
[1] 4.472136
```

If you have existing parameter estimates and want to apply them here, you may have to do a unit conversion. The exact conversion will depend on what you have, but here is an example. Say your observations, z, are the number of individuals per grid cell of size length a. And say that when you estimated the edge function parameters (e_0, D_{max}, k, and maybe D_0), you gave distances in meters rather than number of grid cells. Since D_{max} and D_0 have units of length, the adjusted values you should use with the unit grid cells here are $D'_{max} = D_{max}/a$ and $D'_0 = D_0/a$. Since k already has units of per grid cell, $k' = k$. Since e_0 has units of individuals per length, $e'_0 = e_0 \times a$.

4.1 Input files

Here is an example of an input file for a gridded landscape. Note that `0` signifies non-habitat, any other positive number signifies habitat, and spaces signify borders between cells (this allows habitat codes of more than one digit).

```
0 0 0 0 0 0 0 0 0 0 2 2 2 2
1 1 1 1 1 0 0 0 0 0 0 2 2 2
1 1 1 1 0 0 0 0 0 0 2 2 2 2
1 1 1 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 0 0 0 0 0 0 1 1
```

Let's read that particular habitat file into a matrix:

```
> map <- read.table(paste(prefix, "smallgrid.dat", sep = ""))
> map <- data.matrix(map)
```

I found it easier to visualize the landscape with slightly different formatting. The function `write.grid()` produces a file, whose contents are printed below. You can find the example files near the `prefix` you defined above (replace `inputfiles` with `outputfiles`).

```
> write.grid(map, "outputfiles/smallgrid.map")
```

```
           ....
.....       ...
....       ....
...
...
......      ..
```

The next step is to identify the cells that are edges. This can be slightly slow for a large landscape. You may also want to write those results to a file for visualization/checking.

```
> edges <- find.edges(map)
> write.edges(map, edges, "outputfiles/smallgrid.edge")
```

creates a file that contains:

```
xxxxx    x
     x    x
    x    x
   x      xxxx
   xxx      xx
      x    x
```

4.2 Response prediction from given parameter values

To compute the response value for each cell, we first need to provide parameter values for the plateau point edge effect function.

```
> params <- list(e0 = -0.3, k = 5, D0 = 0, Dmax = 10)
```

Then we can use `grid.effects()` to treat each habitat cell in turn as the focal cell and compute its response value, z. This step can be slow for a large grid and large values of D_{max}.

```
> z <- grid.effects(map, edges, params)
```

The result is a vector with one item per cell, ordered by column (read down column 1, then read down column 2, etc.). These values are the predicted responses for habitat cells, and `NA` for non-habitat cells. If you didn't want to include some cells identified as edges, you could just remove them from `edges` before calling `grid.effects()`.

To print the results in either tabular or graphic form, you have to take some care to get the orientation of the landscape right. For example, here is how to view `z` in the same orientation as the `map`:

```
> matrix(z, ncol = ncol(map))
```

```
         [,1]     [,2]     [,3]     [,4]     [,5]     [,6] [,7] [,8] [,9] [,10]
[1,]       NA       NA       NA       NA       NA       NA   NA   NA   NA    NA
[2,] 2.757224 2.391091 2.064658 1.761261 1.509462       NA   NA   NA   NA    NA
[3,] 2.778329 2.406429 2.046681 1.702641       NA       NA   NA   NA   NA    NA
[4,] 2.867153 2.501380 2.121457       NA       NA       NA   NA   NA   NA    NA
[5,] 3.019753 2.676498 2.302272       NA       NA       NA   NA   NA   NA    NA
[6,] 3.221152 2.925479 2.582877 2.249200 1.953873 1.756154   NA   NA   NA    NA
        [,11]    [,12]    [,13]    [,14]
[1,] 2.005296 2.304855 2.648967 2.989033
[2,]       NA 2.016763 2.371094 2.747569
[3,] 1.555564 1.826897 2.168306 2.565966
[4,]       NA       NA       NA       NA
[5,]       NA       NA       NA       NA
[6,]       NA       NA 2.388755 2.727175
```

And here is how to write a file for `z` that has the same cell layout as the input file.

```
> write.table(matrix(z, nrow = nrow(map)), file = "outputfiles/smallgrid.z",
+     na = "NA", quote = F, sep = " ", row.names = F, col.names = F)
```

If your input map had a variety of habitat codes and you want to extract the values of `z` that go with each, you can do it like this.

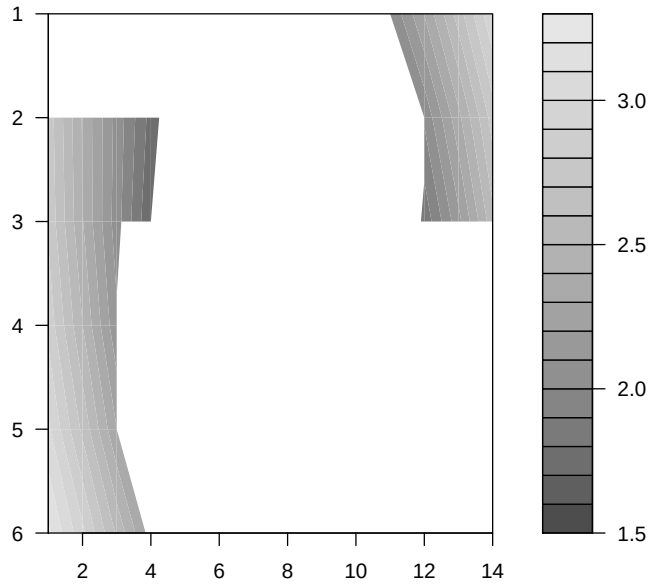
```
> m <- as.vector(as.matrix(map))
> hcodes <- list(one = 1, two = 2)
> sapply(hcodes, f <- function(x) z[m == x])

$one
 [1] 2.757224 2.778329 2.867153 3.019753 3.221152 2.391091 2.406429 2.501380
 [9] 2.676498 2.925479 2.064658 2.046681 2.121457 2.302272 2.582877 1.761261
[17] 1.702641 2.249200 1.509462 1.953873 1.756154 2.388755 2.727175

$two
 [1] 2.005296 1.555564 2.304855 2.016763 1.826897 2.648967 2.371094 2.168306
 [9] 2.989033 2.747569 2.565966
```

To plot the results, turn `z` into a matrix in what seems like the wrong orientation:

```
> z <- matrix(z, byrow = T, ncol = nrow(map))
```

Here's a basic plot of the result.

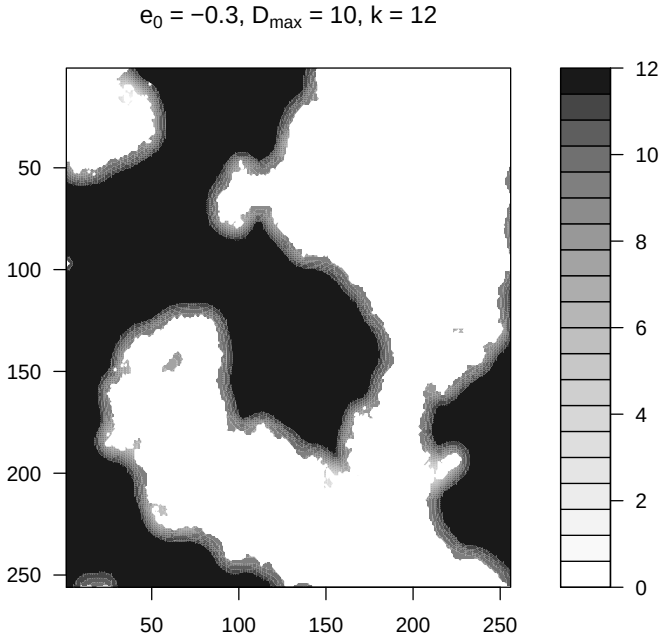
```
> filled.contour(seq(ncol(map)), seq(nrow(map)), z, ylim = c(nrow(map),
+     1), color.palette = gray.colors)
```



(You might instead want to use `contourplot` from the `lattice` package, which uses a clearer formula notation.)

That's not such an exciting landscape. Here's a more elaborate one (the interior areas are black, but that seems that gets lost in the pdf conversion):

```
> map <- read.table(paste(prefix, "biggrid.dat", sep = ""))
> map <- data.matrix(map)
> edges <- find.edges(map)
> params <- list(e0 = -0.3, k = 12, D0 = 0, Dmax = 10)
> z <- grid.effects(map, edges, params)
> z <- matrix(z, byrow = T, ncol = nrow(map))

> label <- substitute(expression(paste(e[0], " = ", e0, ", ", D[max],
+     " = ", Dmax, ", k = ", kval)), list(e0 = params$e0, Dmax = params$Dmax,
+     kval = params$k))
> filled.contour(seq(ncol(map)), seq(nrow(map)), z, ylim = c(nrow(map),
+     1), col = gray.colors(20, 1, 0.1), levels = seq(0, params$k,
+     len = 21), main = eval(label))
```



If the computations are slow and you only want responses predicted for a portion of your landscape, you can create a dataframe of focal cell coordinates and just use those. For example, to predict on just a strip in the upper left corner:

```
> focals <- data.frame(row = rep(seq(5), 2), col = c(rep(1, 5),
+     rep(2, 5)))
> focals

   row col
1    1   1
2    2   1
3    3   1
4    4   1
5    5   1
6    1   2
```

```
7    2   2
8    3   2
9    4   2
10   5   2
```

```
> z <- apply(focals, 1, grid.edge.effect, edges, map, params)
```