

Wildflower writeup

Elizabeth Crone, Mollie Brooks, & Perry de Valpine

July 31, 2012

1 Summary

This is a binomial GLMM with the interesting features of multiple (3) random effects, including two that are crossed.

2 Introduction

These data are from E. Crone and colleagues' long-term study of stages, flowering, and seed pod production of *Astragalus scaphoides*. This model looks at individual flowering as a function of the previous year's stage and seed production (which is 0 if the previous stage was not "Flowering"). With this model, the effect (slope) of previous year's seed production is interpretable as a cost of reproduction. At the population level, the percent of plants flowering shows a strongly oscillating pattern, and experiments have shown that preventing reproduction allows individual plants to flower in successive years.

The purpose of this analysis is to quantify how much the costs differ among individual plants. We estimate this variation using random effects for both the slope and intercept of flowering over individual ID. (Use of random effects is justified because we have very little data for some plants, and because we think a logit-normal distribution is appropriate for variation among individuals.) We also included random effects of year, to account for background variation in flowering probability. Therefore the statistical interest in this project is the ability of different packages to estimate crossed random effects. Of note, some plants never produced seeds, so most or all of the information specifically about individual variation in cost of reproduction comes from a subset of the data.

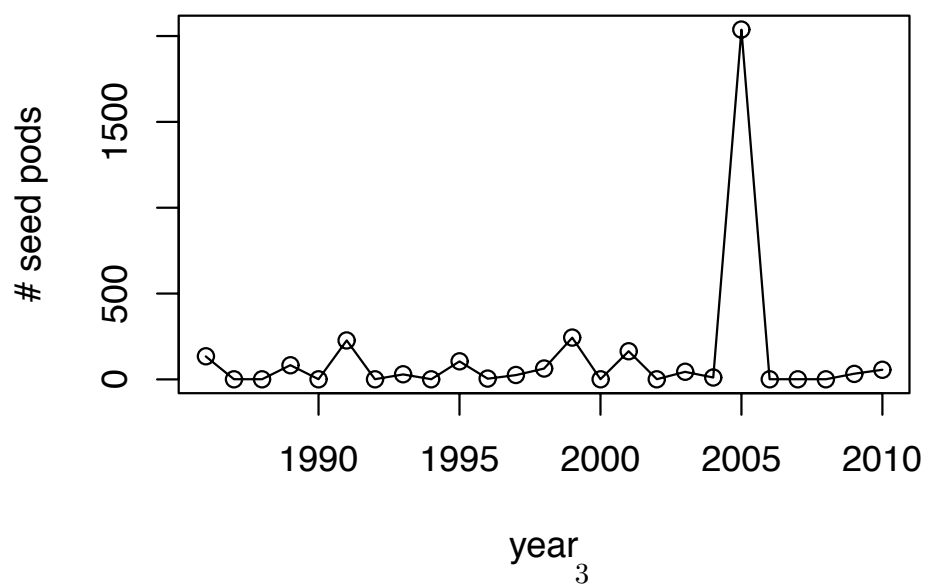
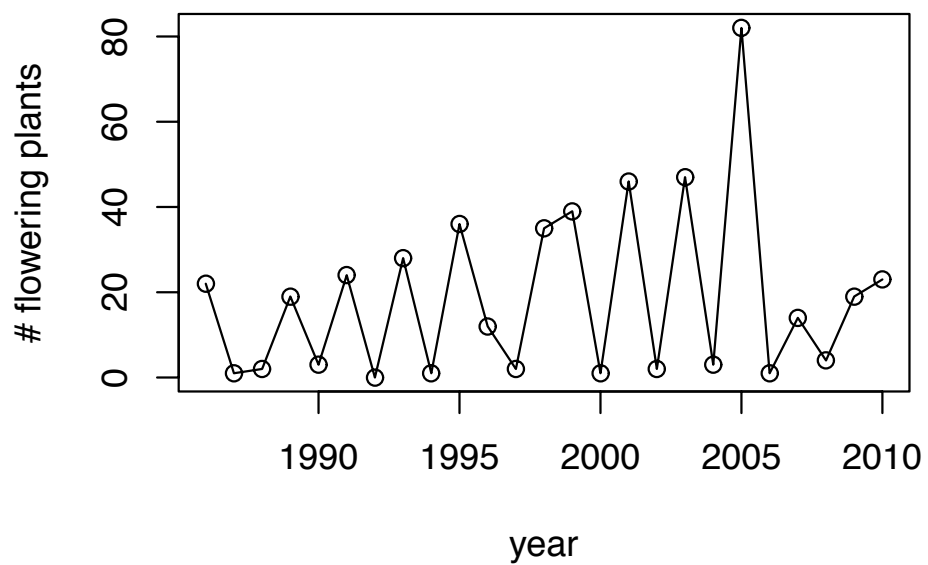
The stage categories are

- D = Dormant
- S = Small
- M = Medium
- L = Large

- F = Flowering (reference category for contrasts)

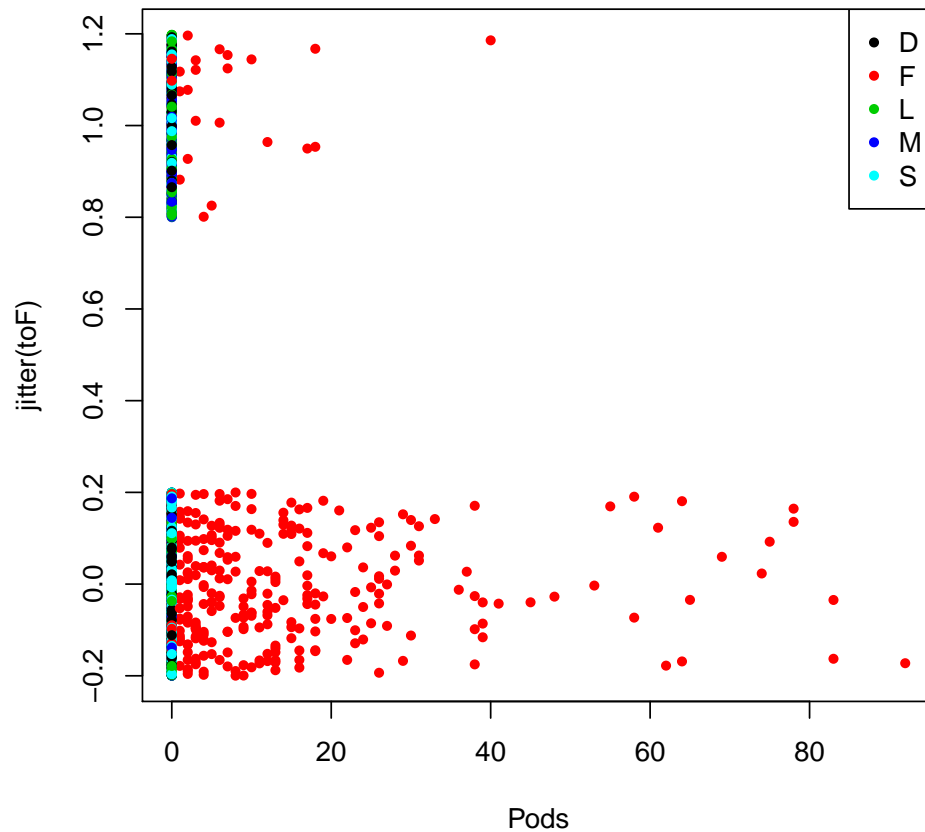
```
> flowers <- read.csv("../DATA/wildflower_pods_complete.csv",  
                      header=TRUE, row.names = 1)
```

The total number of flowering plants and seed pods by year look like:



The relation between flowering and pods in the previous, colored by stage in previous year, looks like:

```
> plot(jitter(toF) ~ Pods, col = State, data = flowers, pch = 20)
> legend(x = "topright", legend = levels(flowers[["State"]]),
        pch = rep(20, 5), col = 1:5)
```



As a preliminary step, lme4 was used to determine a reasonable set of random effects for use in software comparisons. These include a random intercept for years as well as random intercept and slope for individual plants. For the two random effects for plants, there is little support for needing correlation between these, so they are treated as independent. Since plants are crossed with years, two of these three effects are crossed.

3 Basics

This model fits into a GLMM format.

4 R

In R, the package `lme4` was used for fits¹.

```
> library(lme4.0)
```

Set the ordering of the state levels to be biologically decreasing, so "F" is the reference level.

```
> flowers$State <- factor(flowers$State, levels = c("F", "L", "M", "S", "D"))  
  
> fit <- glmer(toF ~ State + Pods + (1 | PlantID) + (1 | Year)  
              + (Pods-1 | PlantID), family = binomial,  
              data = flowers)  
> summary(fit)
```

5 BUGS

Bugs was run using JAGS via R2jags. Several preliminary runs were used to navigate typical types of choices about priors, data transformations, and chain length.

5.1 Data transformations

A common issue for including explanatory variables (covariates) in logistic regression is whether to center (subtract the mean) and standardize (divide by the standard deviation) before using BUGS. Centering often leads to better mixing and standardizing can allow interpretations that compare results for different variables. On the other hand, centering and standardizing require user work to present result on the original scale of biological measurements, if desired. Particularly when interest centers on the standard deviation of the random effects – rather than having these only represent nuisances – transforming results back to the original scale offers a headache for users. Furthermore, we wished to compare results with R (`lme4`) and ADMB, and both of these can achieve fits with the untransformed covariates. We conducted preliminary runs with and without transformation and found that for these data, transformation makes little difference for mixing. Therefore we did not transform.

¹at least temporarily, using the `lme4.0` package on R-forge; this is equivalent to the stable version `lme4` on CRAN

5.2 Priors

For priors, we focused on options for the random effects variances. One option was to use the conjugate prior, which is a gamma distribution for each random effect precision ($1/\text{variance}$). This has the advantage of being conjugate but is not flat and has been questioned in some cases. Another option was to use a uniform prior on each random effect standard deviation, which some researchers believe is more sound. For the uniform prior we used an upper bound of 5, which is far beyond any sampled value, and considered lower bounds of 0.0001 and 0.01. A lower bound is useful because the chain can get sticky at very low values that overall have little support, inhibiting mixing.

Priors for intercepts and the slope parameter were very wide normals ($\text{variance} = 10^6$). Choice of these priors is worth careful consideration because different priors for these – which enter a linear calculation on a logit scale – correspond to different implied priors on the inverse-logit (probability) scale. However, we did not explore other options for these priors as part of this exercise.

We compared the options for standard deviation priors based on three chains run for 10000 iterations, thinned by 10 to record 1000 samples. The Gelman-Rubin statistics for the uniform priors on the standard deviations with a lower bound of 0.0001 were:

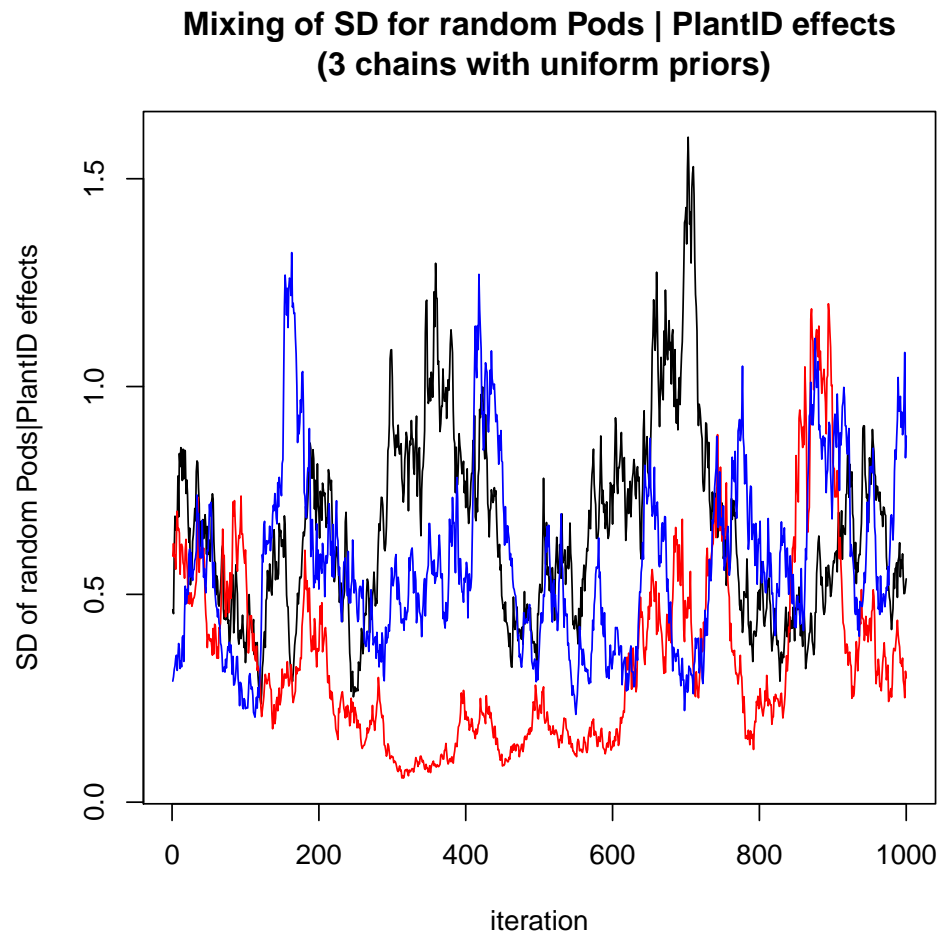
```
> library(R2jags)
> load('../BUGS/test_SDflatprior_10000.Rdata')
> gelman.diag(as.mcmc(wfit_jags_0001boundary_10000))
> ##rm(wfit_jags_0001boundary)
```

The Gelman-Rubin statistics for the gamma priors on the precisions were:

```
> load('../BUGS/test_SDgammaprior_10000.Rdata')
> gelman.diag(as.mcmc(wfit_jags_gamma_priors_10000))
> ##rm(wfit_jags_gamma_priors)
```

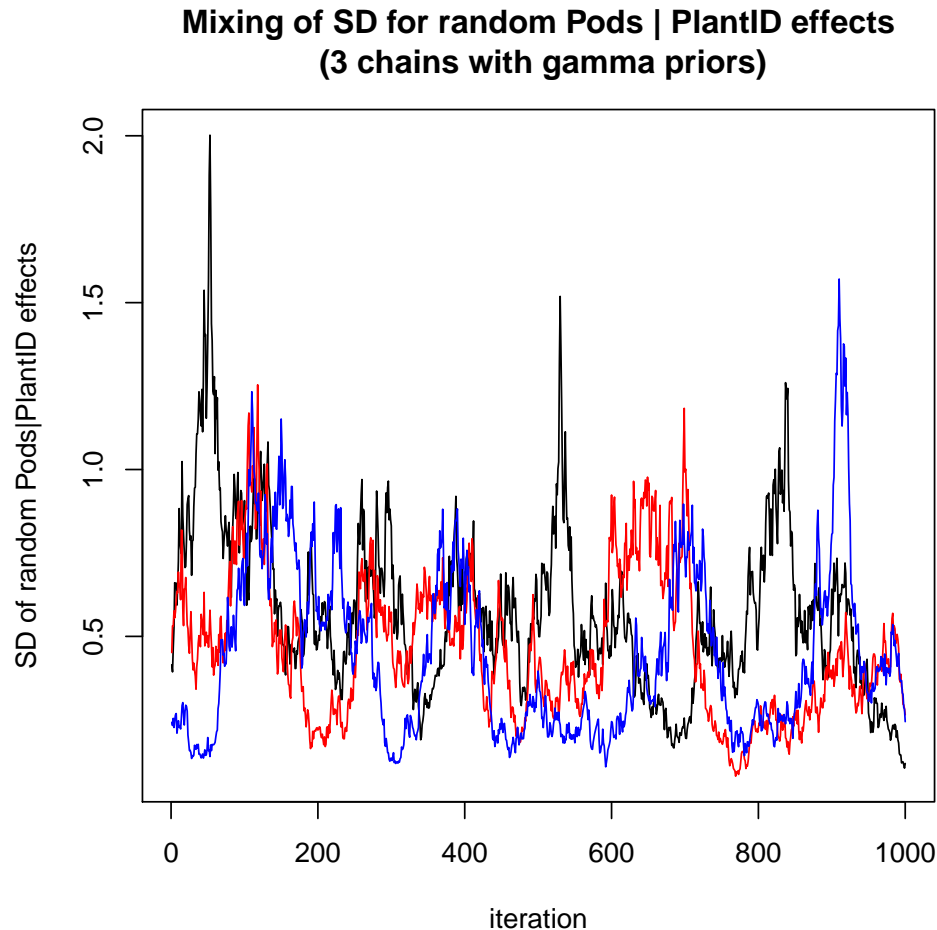
For these sample sizes the gamma prior appears to have better mixing, but we were not fully convinced of this conclusion. Next we illustrate the mixing of these cases. The worst mixing is for the standard deviation of the random slope of Pods for each plant. For the uniform prior it looks like this:

```
> matplot( wfit_jags_0001boundary_10000$BUGSoutput$sims.array[,1:3,'plantSlopeSD'],
           type="l",col=c(1,2,4),lty=1,
           xlab = 'iteration', ylab = 'SD of random Pods|PlantID effects',
           main = 'Mixing of SD for random Pods | PlantID effects\n (3 chains with uniform priors)
```



prior it looks like this:

```
> matplot( wfit_jags_gamma_priors_10000$BUGSoutput$sims.array[,1:3,'plantSlopeSD'],
  type="l",col=c(1,2,4),lty=1,
  xlab = 'iteration', ylab = 'SD of random Pods|PlantID effects',
  main = 'Mixing of SD for random Pods | PlantID effects\n (3 chains with gamma priors)')
while for the gamma
```



While neither of these mixing figures appears to be great, the gamma prior leads to more stickiness near zero. Therefore we choose the uniform prior. In addition we observed from these and other trial runs that there is negligible posterior support below, say, 0.01 for the standard deviations, but mixing stickiness is worse as values move very close to zero. Therefore we made the *ad hoc* choice of 0.01 as a minimum value for the standard deviations. We believe these types of navigation choices and mixing experiments, which can take hours of human time, are not atypical as steps to using BUGS for final results.

Other parameters appear to mix well, but it must be recognized that all dimensions need to mix well in order to trust results. Here is an example of good mixing of the intercept (corresponding to the F(lowering) state). (This sample is from the uniform SD priors):

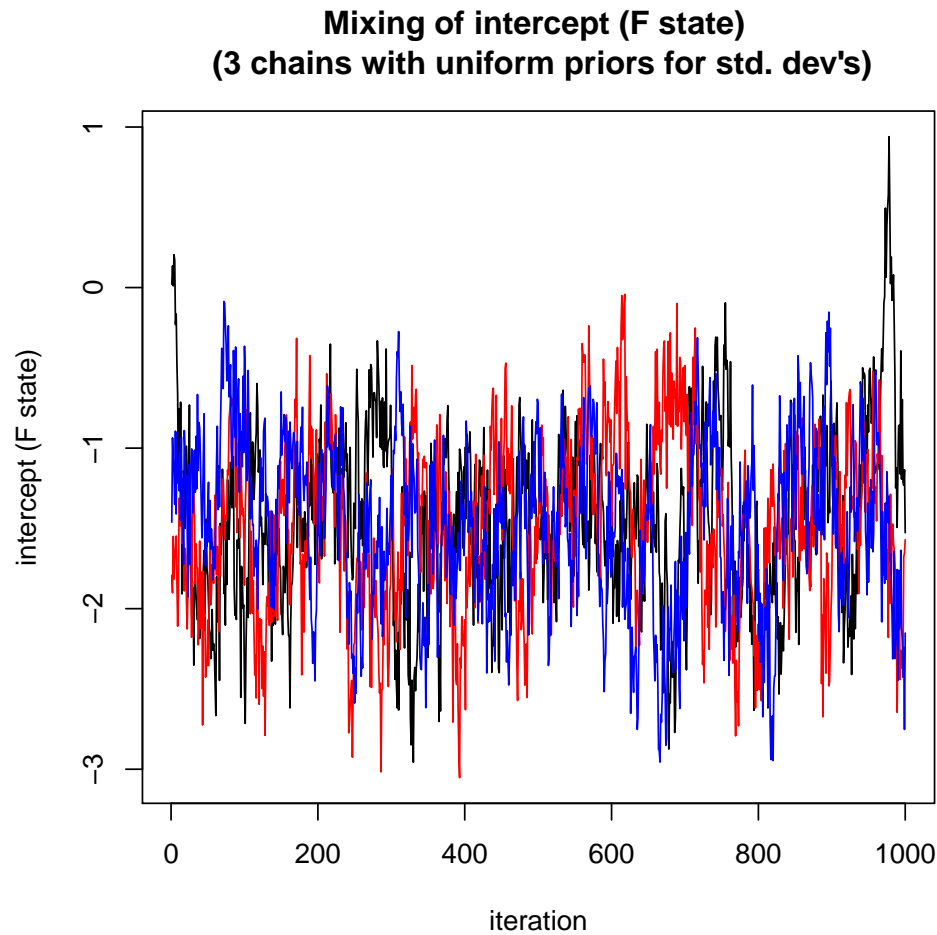
```
> matplot( wfit_jags_gamma_priors_10000$BUGSoutput$sims.array[,1:3,'intercept[1]'],
```



```

type="l",col=c(1,2,4),lty=1,
xlab = 'iteration', ylab = 'intercept (F state)',
main = 'Mixing of intercept (F state) \n (3 chains with uniform priors for std. dev\'s)

```



5.3 Chain length

Finally, with our choice of priors for the random effects standard deviations, we experimented with runs of 1000, 10000, and 100000 iterations to achieve acceptable mixing. For each sample size we used thinning to record an output sample size of 1000. The Gelman-Rubin statistics for these were as follows

```

> load('../BUGS/test_different_run_lengths.Rdata')
> writeLines('1000')
> gelman.diag(as.mcmc(wfit_jags_1000))
> writeLines('10000')
> gelman.diag(as.mcmc(wfit_jags_10000))
> writeLines('100000')
> gelman.diag(as.mcmc(wfit_jags_100000))

```

Interpretation of Gelman-Rubin statistics requires judgment. For 1000 iterations, the large values for plantSlopeSD, moderate values for some others, and large value for the combined multi-variate result are clearly not acceptable. For 10000 samples, many of the point estimates are not too bad (not too far above 1), but the upper C.I. values indicate they could be unacceptable. Considering the plots of the plantSlopeSD samples, we find 10000 samples unacceptable. We decided to use 100000 samples, although for a single analysis of real data for a publication, we would be likely to increase this even more to alleviate any concern about mixing of the plant slope random effect standard deviation (see above figures).

5.4 Final BUGS code

The final BUGS code was as follows:

```

1 model {
2
3
4   # Priors for random effects precisions
5   yearInterceptSD ~ dunif(0.0001, 5)
6   plantInterceptSD ~ dunif(0.0001, 5)
7   plantSlopeSD ~ dunif(0.0001, 5)
8
9   ## When we use gamma priors for the precision, the lines above
10  ## would be replaced by (for example):
11  ## plantSlopeSD <- 1/sqrt(plantSlopePrecision)
12
13  yearInterceptPrecision <- 1/(yearInterceptSD * yearInterceptSD)
14  plantInterceptPrecision <- 1/(plantInterceptSD * plantInterceptSD)
15  plantSlopePrecision <- 1/(plantSlopeSD * plantSlopeSD)
16
17  ## ... and the above lines would be replaced by:
18  ## plantSlopePrecision ~ dgamma(0.0001, 0.0001)
19
20  # Priors for fixed effects (intercepts) of previous stage
21  ## WE COULD CONSIDER OTHER OPTIONS HERE...
22  for (i in 1:Nstage) {

```

```

23 | intercept[i] ~ dnorm(0, 1.0E-6)
24 | }
25 |
26 | # Prior for slope
27 | ## ... AND HERE
28 | slope ~ dnorm(0, 1.0E-6)
29 |
30 | ## random year effects:
31 | for (i in 1:Nyear) {
32 |   yearInterceptEffect[i] ~ dnorm(0, yearInterceptPrecision)
33 | }
34 |
35 | ## random plant effects:
36 | for (i in 1:Nplant) {
37 |   plantInterceptEffect[i] ~ dnorm(0, plantInterceptPrecision)
38 |   plantSlopeEffect[i] ~ dnorm(0, plantSlopePrecision)
39 | }
40 |
41 | ## logit model and data:
42 | for(i in 1:Ndata) {
43 |   logit(probF[i]) <- intercept[ stage[i] ] +
44 |                       yearInterceptEffect[ year[i] ] +
45 |                       plantInterceptEffect[ plant[i] ] +
46 |                       slope * Pods[i] +
47 |                       plantSlopeEffect[ plant[i] ] * Pods[i]
48 |   toF[i] ~ dbern(probF[i])
49 | }
50 | }

```

6 AD Model Builder (via R2admb)

6.1 Data Structure

To implement the model in ADMB, we started by organizing the data in R. We made a design matrix for the fixed effects part of the model. In a first attempt of the model, we also used design matrices for the random effects, but the model ran very slowly (more than 6 minutes), so we abandoned that formulation. Instead, we decided to exploit the separability of the individual observations. The benefit of exploiting the separability is that this makes the Hessian matrix sparse so that inversion is much faster. The command line option `-shess` tells ADMB to use methods specific to a sparse Hessian. To enable this separability, we created index vectors for the random effects, `plant` and `year`, that are used to access the appropriate elements of the random effects vectors for each observation.

```

> setup_wildflower_data <- function(fdat) {
  year <- fdat$Year - min(fdat$Year) + 1
  plant <- as.integer(fdat$PlantID)

  fdat$Year=as.factor(fdat$Year)
  fdat$State =factor(fdat$State, levels = c("F","L","M","S","D"))

  fdesign=model.matrix(~State+Pods, data=fdat)

  list(nobs=nrow(fdat),
       nplants=length(unique(plant)),
       nyears=length(unique(year)),
       nfixed=ncol(fdesign),
       obs=fdat$toF,
       fdesign=fdesign,
       year=year,
       plant=plant,
       pods=fdat$Pods
  )
}
> fdat=read.csv("../DATA/wildflower_pods_complete.csv")
> data4ADMB=setup_wildflower_data(fdat)

```

Then we used R2admb to write the data to wildflowers.dat which ADMB reads in. The elements of the list sent to this function must be in the same order as they appear in the DATA_SECTION of the .tpl file (see below).

```

> library(R2admb)
> write_dat("wildflowers", L=data4ADMB)

```

6.2 Initial Parameter Values

We expected our fixed effects coefficients to be negative because they're on the logit scale. So we thought it would be a good idea to start these parameters at -1. However, these starting values led the optimizer into a region of parameter space that produced NAs. So instead, we started the fixed effects coefficients at 0. We started the three standard deviations of the random effects at 1. All random effects were started at 0.001.

```

> setup_wildflower_params <- function(nfixed, nplants, nyears){
  list(fcoeffs=rep(0,nfixed),
       plant_sigma_intercept=1,

```

```

        plant_sigma_slope=1,
        year_sigma=1,
        rcoeffs_plant_intercept=rep(0.001, nplants),
        rcoeffs_plant_slope=rep(0.001, nplants),
        rcoeffs_year=rep(0.001, nyears)
    )
}
> params4ADMB=setup_wildflower_params(data4ADMB$nfixed, data4ADMB$nplants, data4ADMB$nyears)

    Then we used R2admb to write the initial values to wildflowers.pin
> write_pin("wildflowers", L=params4ADMB)

```

6.3 ADMB code

6.3.1 Data Section

This is where you define data objects to be read in from the .dat file. It is common practice to first define some integers that are later used as the lengths of dimensions of the data. This is so that the same .tpl file can be used with different data sets. Lines 2 through 5 of the code define these types of data: number of observations, number of plants, number of years, and number of fixed effects. When defining vectors, matrices, or arrays, the range of indices (or sometimes a vector of indices) is written in parentheses beside the object name. Here, **obs**, **year**, **plant**, and **Pods** are vectors with indices from 1 to **nobs**. The design matrix for the fixed effect part of the model (line 7 of the code) has rows indexed 1 to **nobs** and columns indexed 1 to **nfixed**. There can be no spaces in the parentheses.

6.3.2 Parameter Section

This section is where parameters are defined. We put all the fixed effects coefficients together in a vector called **fixedcoeffs** so that we can multiply it by the design matrix of predictors for efficient computation; this is more efficient than having all the predictors in separate vectors and the coefficients defined separately. The indices of this vector must match the column indices of the design matrix.

The ADMB code requires parameters for the standard deviations of each of the random effects because **random_effects_vector** objects in ADMB must be distributed as standard normal and transformed to create the desired distribution. The standard deviations of our random effects are defined on lines 14, 15, and 16. We bounded them between 0.001 and 100; these bounds are defined in parentheses beside the definition. The 2 in the last position of the parentheses indicates that these should be optimized in phase 2 (see next section for explanation).

We define a number, **dummy** that will fill in as a place holder for **plant_int_slope** in parts of the code (description below). It will not be optimized over, so there is no "init_" in its definition.

Our random effects vectors are defined on lines 18,19,and 20. These definitions have to come after all the other parameter definitions. Random effects vectors do get initialized from the pin file, but their definition doesn't require "init_" in front. According to our model, each plant will have some deviation from the overall intercept and slope; so `plant_int` and `plant_slope` have elements for each plant, indexed 1 to `nplants`. Similarly, each year deviates from the overall intercept; so `year_int` has elements for each year, indexed from 1 to `nyears`. Again, the 2 in the last position of the parentheses indicates that these should be optimized in phase 2 (see next section for explanation).

The last definition is the `objective_function_value`; we've called it `jnl1` to stand for "joint negative log-likelihood" which is what we calculate in the `PROCEDURE_SECTION` and store in this object.

6.3.3 Phases

A useful feature of ADMB is that it can optimize over a subset of the parameters and later bring in additional parameters to be optimized in additional phases. It is common to first optimize over the fixed effects of a model and then bring in the random effects in subsequent phases. That's what we did here. We found that without using this feature, the model didn't converge. Using this feature is simple; any parameter that you want to optimize in a phase other than the first one, put the number of that phase as the last number in the parentheses where you define that parameter in the `PARAMETER_SECTION`. To optimize the random effects in phase 2, we placed a 2 in the parentheses beside `plant_int_std`, `plant_slope_std`, `year_int_std`, `plant_int`, `plant_slope`, and `year_int`. These are on lines 14,15,16,18,19,and 20 of the code below.

6.3.4 Procedure Section

This section is where we calculate the joint negative log-likelihood by adding up the log-likelihoods of the binomially distributed observations and the standard normally distributed random effects. It's important to initialize `jnl1` (line 25) because we're adding to it. The first for loop adds the negative log-likelihood of the observations to `jnl1` (lines 26 to 36), the second for loop adds the negative log-likelihood of the random effects of plant on slope and intercept (lines 37 to 41), and the third loop adds the negative log-likelihood of the random effects of year on the intercept (lines 42 to 45).

The point of using separable functions is to make the Hessian as sparse as possible, so that efficient algorithms can be used. For ADMB to know how sparse the Hessian is, only send parameters to the separable function that must be used together for a particular observation. The if-else conditional statement on lines 28 to 35 is there to make the Hessian as sparse as possible to make the computation more efficient. We avoid unnecessarily sending the random effect for `plant_slope` when there are zero pods, because this term of the predictor equation is automatically zero.

1 | DATA_SECTION

```

2      init_int nobs;
3      init_int nplants;
4      init_int nyears;
5      init_int nfixed;
6      init_vector obs(1,nobs);
7      init_matrix fdesign(1,nobs,1,nfixed);
8      init_ivector year(1,nobs);
9      init_ivector plant(1,nobs);
10     init_ivector pods(1,nobs);
11
12     PARAMETER_SECTION
13         init_vector fixedcoeffs(1,nfixed);
14         init_bounded_number plant_int_std(0.001,100,2);
15         init_bounded_number plant_slope_std(0.001,100,2);
16         init_bounded_number year_int_std(0.001,100,2);
17         number dummy;
18         random_effects_vector plant_int(1,nplants,2);
19         random_effects_vector plant_slope(1,nplants,2);
20         random_effects_vector year_int(1,nyears,2);
21         objective_function_value jnll;
22
23     PROCEDURE_SECTION
24         dummy=0.0;
25         jnll=0.0;
26         for(int i=1; i<=nobs; i++)
27         {
28             if(pods(i)==0)//don't bother with the random effect on slope,
29                 send the dummy
30             {
31                 binom(i, plant_int(plant(i)), dummy, year_int(year(i))
32                     ,fixedcoeffs, plant_int_std, plant_slope_std,
33                     year_int_std);
34             }
35             else
36             {
37                 binom(i, plant_int(plant(i)), plant_slope(plant(i)),
38                     year_int(year(i)), fixedcoeffs, plant_int_std,
39                     plant_slope_std, year_int_std);
40             }
41         }
42         for(int pl=1; pl<=nplants; pl++)
43         {
44             rand(plant_slope(pl));
45             rand(plant_int(pl));
46         }

```

```

42     for(int y=1; y<=nyears; y++)
43     {
44         rand(year_int(y));
45     }
46
47 SEPARABLE_FUNCTION void binom(int i, const dvariable& plant_int, const
    dvariable& plant_slope, const dvariable& year_int, const dvar_vector&
    fixedcoeffs, const dvariable& plant_int_std, const dvariable&
    plant_slope_std, const dvariable& year_int_std)
48     dvariable logitp=fdesign(i)*fixedcoeffs+
49         plant_int*plant_int_std+
50         plant_slope*pods(i)*plant_slope_std+
51         year_int*year_int_std;
52     dvariable p=1.0/(1.0+exp(-logitp));
53     jnll -=obs(i)*log(p); //flowering
54     jnll -= (1.0-obs(i))*log(1.0-p); //not flowering
55
56 SEPARABLE_FUNCTION void rand(const dvariable& ui)
57     jnll +=0.5*(ui*ui)+0.5*log(2.*M_PI);
58
59 TOP_OF_MAIN_SECTION
60     arrmbsize = 30000000;
61     gradient_structure::set_MAX_NVAR_OFFSET(20000000);
62     gradient_structure::set_CMPDIF_BUFFER_SIZE(80000000);
63     gradient_structure::set_GRADSTACK_BUFFER_SIZE(8000000);

```

6.4 Running the code in R via R2admb and getting results

Load the package and let it get its bearings.

```

> library(R2admb)
> setup_admb()

```

```
[1] "/usr/local/admb"
```

First we compile the code to make the executable

```
> compile_admb("wildflowers", safe=TRUE, re=TRUE, verbose=TRUE)
```

Then we can run the executable

```
> run_admb("wildflowers", verbose=FALSE, extra.args="-shess -noinit -nox")
```


The extra argument `"-shess"` tells ADMB that the Hessian will be sparse and to use algorithms that efficiently operate on this type of matrix. The extra argument `"-noinit"` tells ADMB to start the random effects from the last optimum values, instead of the pin file values, when doing the Laplace approximation. `"nox"` reduces the amount of information output while it's running.

Then we can read in the results and view them

```
> fit_admb <- read_admb("wildflowers")
```

The whole summary contains all of the random effect estimates for all years and plants so we'll only output the first 9 coefficients which are the main ones of interest.

```
> coef(summary(fit_admb))[1:9,]
```

	Estimate	Std. Error	z value	Pr(> z)
fixedcoeffs1	-1.4730500	0.566550	-2.6000353	9.321417e-03
fixedcoeffs2	-0.1233010	0.250570	-0.4920821	6.226613e-01
fixedcoeffs3	-0.8442930	0.246020	-3.4318064	5.995755e-04
fixedcoeffs4	-2.4744900	0.259610	-9.5315666	1.549274e-21
fixedcoeffs5	-2.2434400	0.269840	-8.3139638	9.255737e-17
fixedcoeffs6	-0.1331380	0.080221	-1.6596402	9.698685e-02
plant_int_std	0.6785570	0.107020	6.3404687	2.290672e-10
plant_slope_std	0.1687589	0.092847	1.8176018	6.912502e-02
year_int_std	2.3035323	0.409310	5.6278426	1.824775e-08

6.5 MCMC

To get confidence intervals on estimates of variance, it's best to do MCMC sampling. This can be done in ADMB. The default is to have uniform priors. We need to add an sdreport declaration to the tpl file's `PARAMETER_SECTION`

```
sdreport_number plant_int_std2;
```

and at the bottom of the `PROCEDURE_SECTION` store the appropriate value in that object

```
plant_int_std2=plant_int_std;
```

Then to run it and see the results, we would use the following code

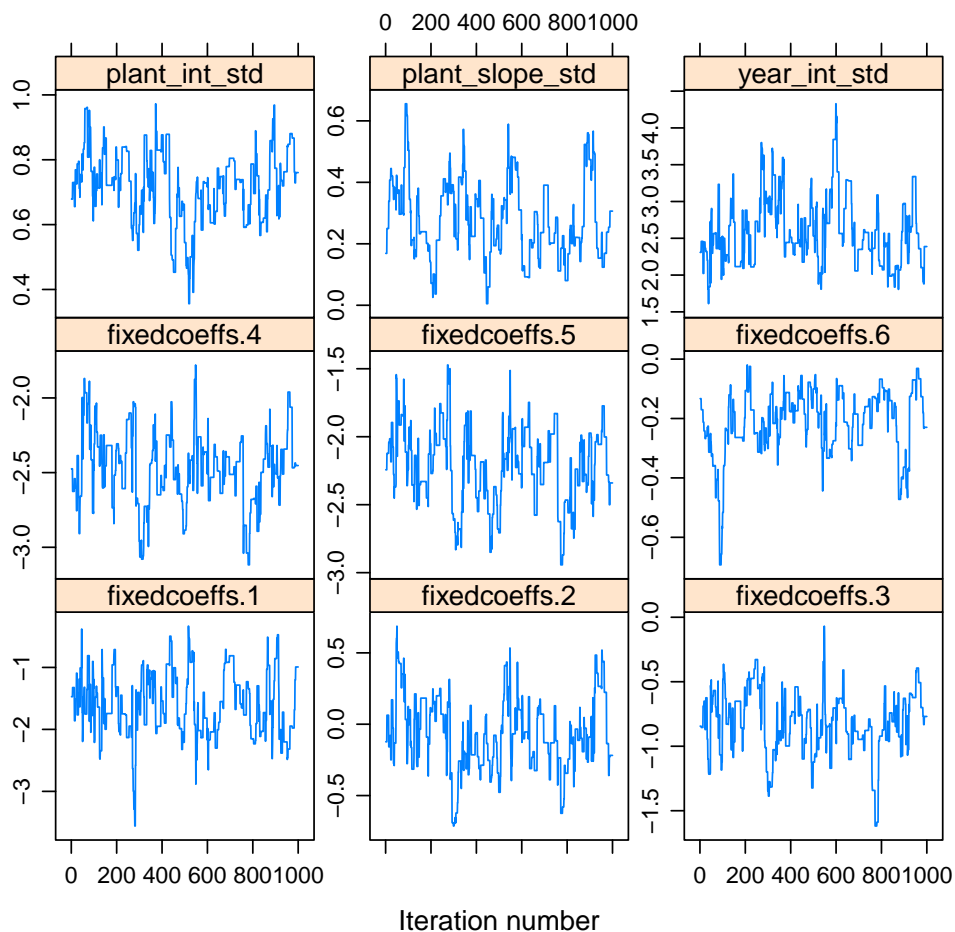
```
> compile_admb("wildflowers_mcmc", safe=TRUE, re=TRUE, verbose=TRUE)
> run_admb("wildflowers_mcmc", verbose=TRUE, extra.args="-shess -noinit -nox",
  mcmc=TRUE, mcmc.opts=mcmc.control(mcmc=50000,
  mcmcpars=c("plant_int_std", "plant_slope_std", "year_int_std")))
> fit_mcmc <- read_admb("wildflowers_mcmc", mcmc=TRUE)
```

Then we can look at the effective sample size and the trace of the parameters of interest to see if our chain ran OK:

```
> library(coda)
> mmc <- as.mcmc(fit_mcmc$mcmc)
> effectiveSize(mmc[,1:9])
```

fixedcoeffs.1	fixedcoeffs.2	fixedcoeffs.3	fixedcoeffs.4	fixedcoeffs.5
40.15309	32.64358	33.72549	33.88444	33.47096
fixedcoeffs.6	plant_int_std	plant_slope_std	year_int_std	
20.38139	27.31991	20.43794	34.17207	

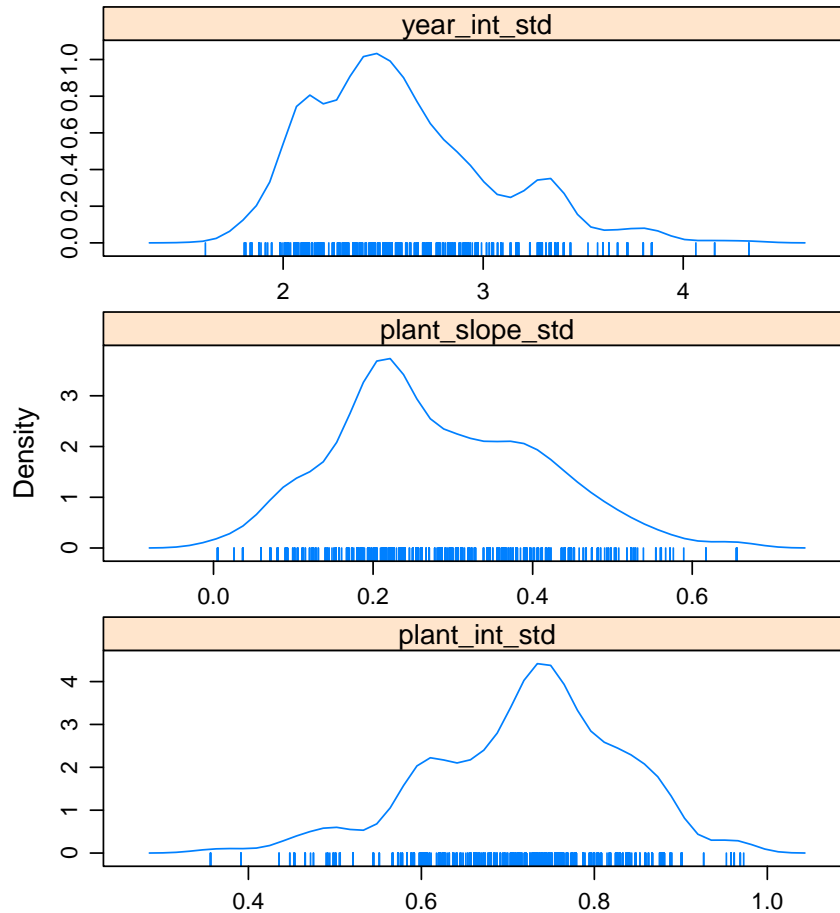
```
> print(xyplot(mmc[,1:9],layout=c(3,3),asp="fill"))
```



The trace looks ok

so we want to look at the density of the standard deviation parameters

```
> print(densityplot(mmc[,7:9]))
```



7 Postscript: Confidence Limits in lmer

lme4 does not have built-in routines for generating confidence limits of random-effects variance estimates for GLMMs. The only way to generate confidence limits for fixed effects is by using the coefficient standard errors. We evaluated two ways of generating confidence limits by bootstrapping: (1) parametric bootstrapping, i.e., evaluating the distribution across bootstrap data sets generated using the `simulate()` function in `lmer`; (2) nonparametric bootstrapping, i.e., evaluating the distribution across bootstrap data sets generated by resampling the original data frame with replacement. (In this case, we did not stratify sampling by `PlantID` or by `year`.) Code for one

parametric bootstrap replicate (calling a model called `fit`, assumes `lme4` is loaded, `fruitdat1` is the original data, and output created to hold coefficients):

```
> tmp.fit = refit(fit,simulate(fit)$sim_1)
```

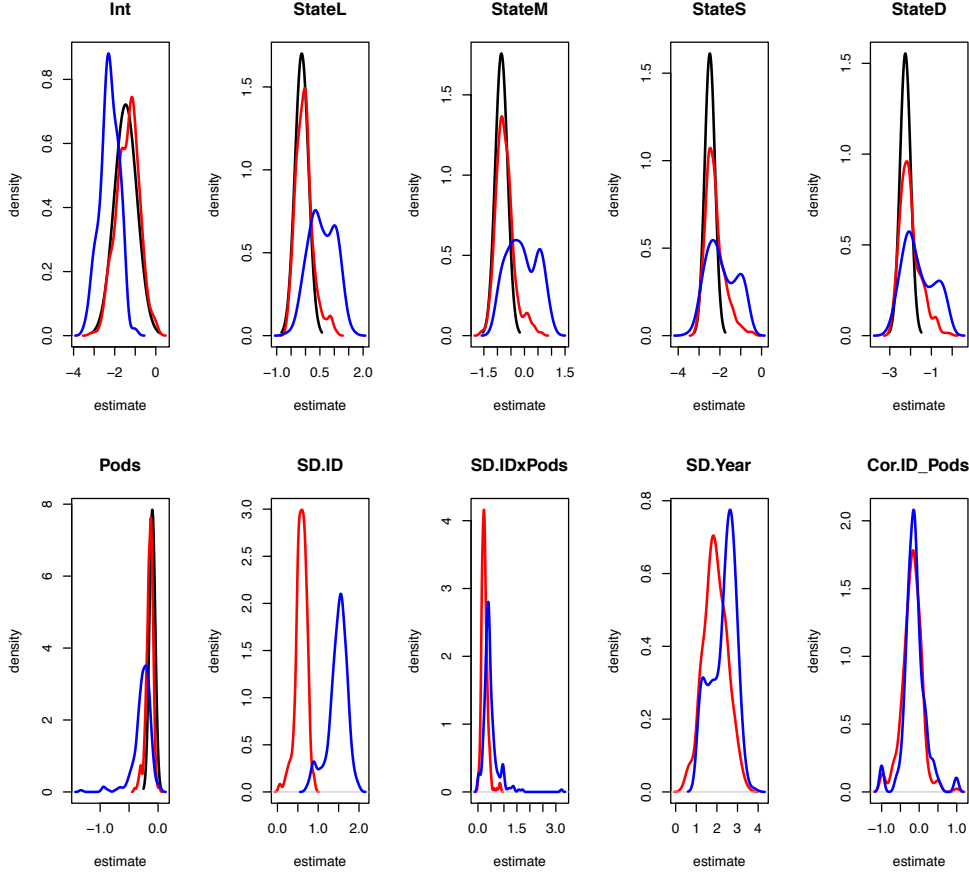
Note that using `refit` is faster than refitting the model from scratch or using `update`.

```
> getcoefs = function(fit) {  
  c(coef(summary(fit))[,1],  
    unlist(sapply(VarCorr(fit),attr,"stddev")))  
  ## attr(VarCorr(fit)[["PlantID"]], "correlation")[1,2])  
}  
> coef.tmp = getcoefs(tmp.fit)
```

Code for one nonparametric bootstrap replicate (same assumptions as above):

```
> dat.tmp2 = flowers[sample(nrow(flowers), replace = TRUE),]  
> fit.tmp2 = refit(fit,dat.tmp2$toF)  
> coef.tmp2 = getcoefs(fit.tmp2)
```

Here are the results of comparing these three methods: BLACK lines are the distribution defined by means and standard errors for fixed effects, RED lines are the distribution of parametric bootstrap samples (250 replicates), and BLUE lines are the distribution of nonparametric bootstrap samples (250 replicates):



Overall, parametric bootstrapping led to distributions that were similar to, but slightly wider than, asymptotic distributions. I (EEC) suspect these are the best measure of confidence in parameter estimates, though we are not aware of a general proof of this assertion. Nonparametric bootstrapping did not work well for these data. I (EEC) suspect poor performance of parametric bootstrapping results from the very sparse nature of the data; some replicate samples might not include any plants with high seed set (see graph of flowering vs. pods). We did not include calculation of confidence limits in the analyses of simulated data because the time demands were prohibitive. On my laptop, 10 replicates of parametric bootstrapping took 525 seconds to run.