Lab 6

Ben Bolker, modified at several places by Bob Douma and Alejandro Morales. Bayesian part by Alejandro Morales Sierra.

19 October 2018

Learning goals

You will learn how to:

1. Program the likelihood function of a model.
2. Estimate the parameters of a model through maximum likelihood.
3. Estimate the confidence intervals of the model parameters through profiling and the quadratic approximation.
4. Estimate parameters in a Bayesian framework and how parameter uncertainty can be assessed (time permitting and when of interest to you).

Fitting models to data

In this exercise you will learn how to fit models to data through means of maximum likelihood and compare the likelihood of different models (hypotheses). Fitting a model to data through likelihood requires that you take four steps:

1. Specify how the dependent variable depends on the independent variable, i.e. specify a function how the mean of y depends on the value of x.
2. Specify a probability distribution to describe the deviations of the observations from the mean
3. Choose the parameters of the deterministic model and the probability model such that the negative log likelihood is lowest.
4. Compare the likelihood of alternative models (change the deterministic function or the stochastic function) and compare with AIC(c) or BIC which model is most parsimonious.

To fit a model through means of maximum likelihood you need to specify a function that calculate the negative log likelihood (NLL) based on the data and the parameter values. For example to calculate the NLL of a linear model and a normal distribution the following function works:

```
nll = function(par,y,x){
  a = par[1]
  b = par[2]
  sd = par[3]
  # this calculates the mean y for a given value of x: the deterministic function
  mu = a+b*x
  # this calculates the likelihood of the function given the probability
  # distribution, the data and mu and sd
  nll = -sum(dnorm(y,mean=mu,sd=sd,log=T))
  return(nll)
}
```

Note that `-sum(log(dnorm(y,mean=mu,sd=sd)))` should not be used as it may lead to underflow (the computer cannot store very very small probabilities) and therefore to optimisation problems.

Next we specify a function to find the maximum likelihood estimate

```
par=c(a=1,b=1,c=1) # initial parameters
opt1 = optim(par=par,nll,x=x,y=y) # y represents the data, x the independent variable
```

It can also be done through `mle2`

```
nll.mle = function(a,b,sd){
  # this calculates the mean y for a given value of x: the deterministic function
  mu = a+b*x
  # this calculates the likelihood of the function given the probability
  # distribution, the data and mu and sd
  nll = -sum(dnorm(y,mean=mu,sd=sd,log=T))
  return(nll)
}
```

```
# the data should be supplied through data and the parameters through list().
mle2.1 = mle2(nll.mle,start=list(a=1,b=1,sd=1),data=data.frame(x,y))
summary(mle2.1)
```

Made-up data: negative binomial

The simplest thing to do to convince yourself that your attempts to estimate parameters are working is to simulate the "data" yourself and see if you get close to the right answers back.

Start by making up some negative binomial ‘`data'': first, set the random-number seed so we get consistent results across different`R‘ sessions:

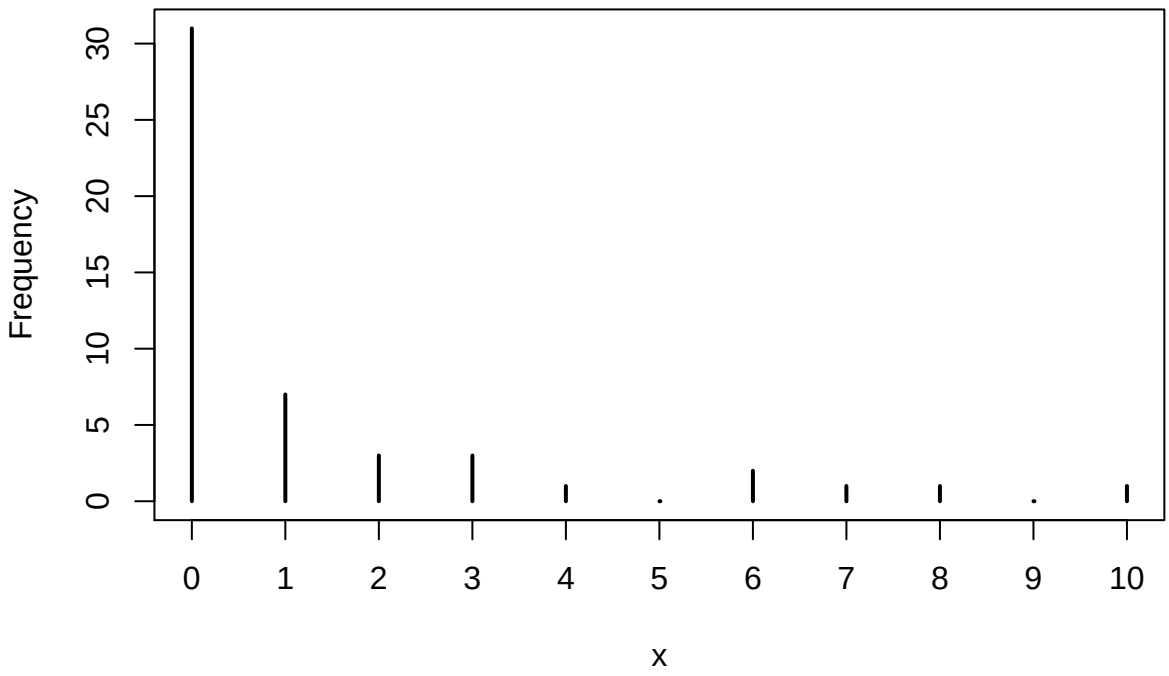
```
set.seed(1001)
```

Generate 50 values with $\mu = 1$, $k = 0.4$ (save the values in variables in case we want to use them again later, or change the parameters and run the code again):

```
mu.true=1
k.true=0.4
x = rnbinom(50,mu=mu.true,size=k.true)
```

Take a quick look at what we got:

```
plot(table(factor(x,levels=0:max(x))),
     ylab="Frequency",xlab="x")
```



(reminder: I won't always draw the pictures, but it's good to make a habitat of examining your variables (with `summary()` etc. and graphically) as you go along to make sure you know what's going on!)

Negative log-likelihood function for a simple draw from a negative binomial distribution: the first parameter, `p`, will be the vector of parameters, and the second parameter, `dat`, will be the data vector (in case we want to try this again later with a different set of data; by default we'll set it to the 'x} vector we just drew randomly).

```
NLLfun1 = function(p,dat=x) {
  mu=p[1]
  k=p[2]
  -sum(dnbinom(x,mu=mu,size=k,log=TRUE))
}
```

Calculate the negative log-likelihood for the true values. I have to combine these values into a vector with `c()` to pass them to the negative log-likelihood function. Naming the elements in the vector is optional but will help keep things clear as we go along:

```
nll.true=NLLfun1(c(mu=mu.true,k=k.true)); nll.true
```

The NLL for other parameter values that I know are way off ($\mu = 10$, $k = 10$):

```
NLLfun1(c(mu=10,k=10))
```

```
## [1] 291.4351
```

Much higher negative log-likelihood, as it should be.

Find the method-of-moments estimates for μ and k:

```
m = mean(x)
v = var(x)
mu.mom = m
k.mom = m/(v/m-1)
```

Negative log-likelihood estimate for method of moments parameters:

```
nll.mom=NLLfun1(c(mu=mu.mom,k=k.mom)); nll.mom
```

```
## [1] 72.08996
```

Despite the known bias, this estimate is better (lower negative log-likelihood) than the "true" parameter values. The Likelihood Ratio Test would say, however, that the difference in likelihoods would have to be greater than $\chi^2_2(0.95)/2$ (two degrees of freedom because we are allowing both μ and k to change):

```
ldiff=nll.true-nll.mom;ldiff
```

```
## [1] 0.5576733
```

```
qchisq(0.95,df=2)/2
```

```
## [1] 2.995732
```

So — better, but not significantly better at $p = 0.05$. `pchisq(2*ldiff,df=2,lower.tail=FALSE)` would tell us the exact p-value if we wanted to know.)

But what is the MLE? Use `optim` with the default options (Nelder-Mead simplex method) and the method-of-moments estimates as the starting estimates (`par`):

```
O1 = optim(fn=NLLfun1,par=c(mu=mu.mom,k=k.mom),hessian=TRUE);
O1
```

```
## $par
##        mu         k
## 1.2602356 0.2884793
##
## $value
## [1] 71.79646
##
## $counts
## function gradient
##       45       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##             mu            k
## mu 7.387808317  0.004901576
## k  0.004901576 97.372581394
```

The optimization result is a list with elements:

- the best-fit parameters (`O1$par`, with parameter names because we named the elements of the starting vector—see how useful this is?);}
- the minimum negative log-likelihood (`O1$value`);

- information on the number of function evaluations (`O1$counts`; the `gradient` part is `NA` because we didn't specify a function to calculate the derivatives (and the Nelder-Mead algorithm wouldn't have used them anyway)
- information on whether the algorithm thinks it found a good answer `O1$convergence`, which is zero if `R` thinks everything worked and uses various numeric codes (see `?optim` for details) if something goes wrong;
- `O1$message` which may give further information about the when the fit converged or how it failed to converge;
- because we set `hessian=TRUE`, we also get `O1$hessian`, which gives the (finite difference approximation of) the second derivatives evaluated at the MLE.

The minimum negative log-likelihood (`round(O1$value,2)`) is better than either the negative log-likelihood corresponding to the method-of-moments parameters (`round(nll.mom,2)`) or the true parameters (`round(nll.true,2)`), but all of these are within the LRT cutoff.

Now let's find the likelihood surface, the profiles, and the confidence intervals.

The likelihood surface is straightforward: set up vectors of μ and k values and run `for` loops, set up a matrix to hold the results, and run `for` loops to calculate and store the values. Let's try μ from 0.4 to 3 in steps of 0.05 and k from 0.01 to 0.7 in steps of 0.01. (I initially had the μ vector from 0.1 to 2.0 but revised it after seeing the contour plot below.)

```
muvec = seq(0.4,3,by=0.05)
kvec = seq(0.01,0.7,by=0.01)
```

The matrix for the results will have rows corresponding to μ and columns corresponding to k:

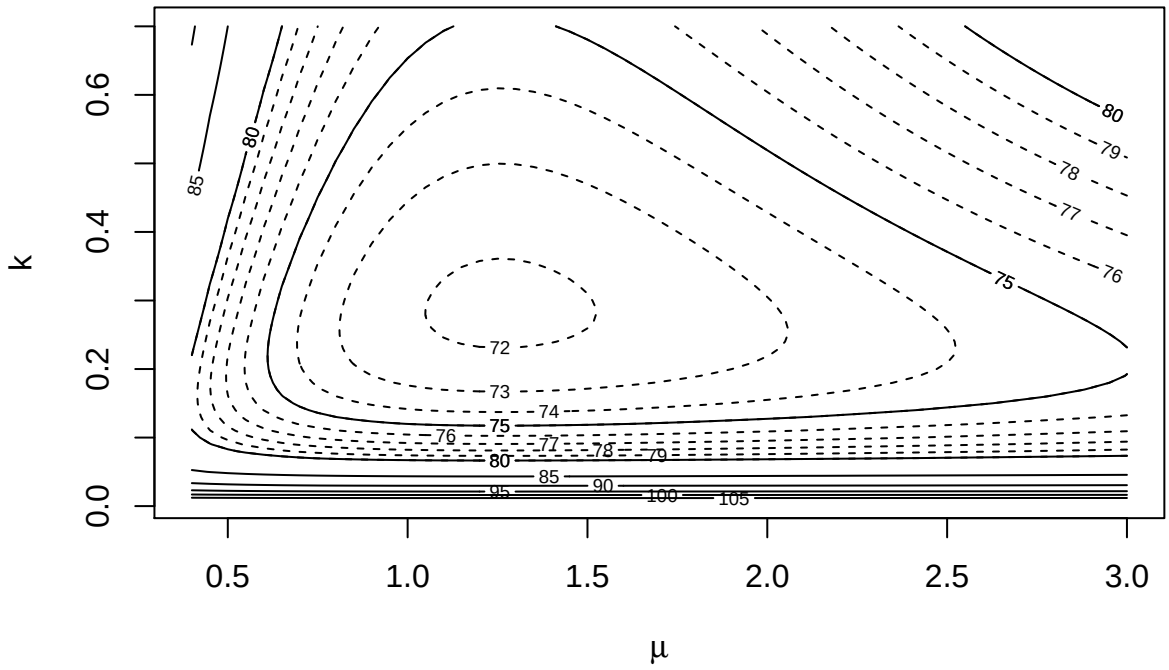
```
resmat = matrix(nrow=length(muvec),ncol=length(kvec))
```

Run the `for` loops:

```
for (i in 1:length(muvec)) {
  for (j in 1:length(kvec)) {
    resmat[i,j] = NLLfun1(c(muvec[i],kvec[j]))
  }
}
```

Drawing a contour: the initial default choice of contours doesn't give us fine enough resolution (it picks contours spaced 5 apart to cover the range of the values in the matrix), so I added levels spaced 1 log-likelihood unit apart from 70 to 80 by doing a second `contour` plot with `add=TRUE`.

```
contour(muvec,kvec,resmat,xlab=expression(mu),ylab="k")
contour(muvec,kvec,resmat,levels=70:80,lty=2,add=TRUE)
```



Maximum likelihood and covariates

The following steps will lead you through the model fitting procedure when we have a covariate.

1. Take the first dataset (or another one from the six that we have been working with before) and tweak the above functions such that it matches with the deterministic and stochastic model that you have chosen. In case you got stuck in the previous exercises where you had to choose a deterministic function and a stochastic function see next page for suggestions.

   hint: In a previous exercise you have eyeballed the parameter values of the functions, you can use these as starting values.

   hint: In case you get convergence problems, further adapt your starting values, or choose a different optimizer. For example Nelder-Mead is a robust one, e.g. `method = "Nelder-Mead"`.

2. Change the determinstic function for a possible alternative determinstic function
3. Compare the likelihoods of the data given both models
4. Apply model selection criteria and conclude which model fits that data best.
5. Does the model makes sense from a biological perspective?

Optional and time permitting:
6. Repeat the above procedure for the other 5 datasets

Optimisation problems and assessing the confidence limits of parameter estimates

Fitting a model to data requires you to specify a relationship between variables. After specifying this relationship we need to fit parameters of this model that best fits the data. This fitting is done through computer algorithms (optimizers). However, sometimes it may be hard to fit a model to data. After having found the best fitting model, you want to assess how certain you are about the parameter estimates. For assessing the uncertainty of model parameters several methods exist that have pros and cons.

This exercise has two purposes. First you will learn that an innocent looking function can be challenging to fit. Second, you will learn to assess the uncertainty in the parameter values. For assessing the uncertainty in the parameter estimates there are two methods: the profiling method and the quadratic approximation. Bolker recommends to use the likelihood profile for assessing the uncertainty in the parameters because this one is more accurate than the one based on the Hessian matrix.

1. Take the first dataset of the six datasets you have worked with earlier on. Assume that the function was generated by the monomolecular function $a(1 - e^{(-bx)}$. Fit this model with normally distributed errors through this data with `mle2` and optim method `Nelder-Mead`. Choose four different starting points of the optimisation: `start_a = c(5,10,20,30)`, `start_b = c(0.001,0.005,0.01,0.1)` and compare the NLL of those four optimisations. Plot the curves into the plot with data and try to understand what happened.

2. To understand the behaviour of the optimisation routine we will plot the likelihood surface over a range of values of a and b. For a choose a number of parameter values in the range of 0-40 and for b choose a number of values in the range 0.1-10. Calculate for each combination the NLL and plot the NLL surface using `contour` plot. For more insight into the functioning of what the optimisation method did, you can add the starting points that you gave to mle2 and the best fitting points, use `points()` for this. Do you have a clue why the optimisation did not find the minimum point in the landscape? Now zoom in and choose values for b in the range of 0.001-0.03 and check again the NLL surface.

   hint: See Bolker Lab 6 for inspiration on coding.

   hint: You can use a for a double for-loop to run over all parameters

   hint: Store the NLL results in a matrix (you can make a 100x100 matrix by `matrix(NA,nrow=100,ncol=100)`).

3. Calculate the confidence intervals of the parameters through constructing the likelihood profile. Consult page 106 of Bokler or Lab 6 for how to calculate the confidence intervals based on the likelihood profile. Use the following pseudocode to achieve this:

   a. Adapt the likelihood function such that one parameter is not optimised but chosen by you, say parameter a.
   b. Vary a of a range and optimise the other parameteters.
   c. Plot the NLL as a function of parameter a.
   d. Find the values of a that enclose $-L + \chi^2(1 - \alpha)/2$. In `R` this can be done through `qchisq(0.95,1)/2`.
   e. Compare your results with the results from the `R` function `confint()`. `confint()` uses the profiling method along with interpolation methods.

4. *(time permitting)* Calculate the confidence intervals through the quadratic approximation. Take the following steps to achieve this:

   a. Get the standard error of the parameter estimates through `vcov`. Note that `vcov` return the variance/covariance matrix
   b. Calculate the interval based on the fact that the 95% limits are 1.96 (qnorm(0.975,0,1)) standard deviation units away from the mean.

5. *(time permitting)* Plot the confidence limits of the both method and compare the results. Is there a big difference between the methods?

6. To assess the uncertainty in the predictions from the model you can construct population prediction intervals (PPIs, see 7.5.3 Bolker). Population prediction intervals shows the interval in which a new observation will likely fall. To construct the PPI take the following steps

   a. Simulate a number of parameter values taken the uncertainty in the parameter estimates into account.

      hint: If the fitted mle object is called `mle2.obj`, then you can extract the variance-covariance matrix by using `vcov(mle2.obj)`. You can extract the mean parameter estimates by using `coef(mle2.obj)`. Now you are ready to simulate 1000 combinations of parameter values through `z = mvrnorm(1000,mu=coef(mle2.obj),Sigma=vcov(mle2.obj))`. `mvrnorm` is a function to randomly draw values from a multivariate normal distribution.

   b. Predict the mean response based on the simulated parameter values and the values of x

      hint: make a for-loop and predict for each simulated pair of parameter values the mean for a given x. Thus `mu = z[i,1]*(1-exp(-z[i,2]*x))`

   c. Draw from a normal distribution with a mean that was predicted in the previous step and the sd that you simulated in step a.

      hint: `pred = rnorm(length(mu),mean=mu,sd=z[i,3])`. Store pred in a matrix with each simulated dataset in a seperate row.

   d. Calculate for each value of x the 2.5% and the 97.5% quantiles

      hint: If the predictions are stored in a matrix `mat`, you can use `apply(mat,2,quantile,0.975)` to get the upper limit.

Bayesian parameter estimation: negative binomial

From Bayes rule to log posterior

The aim of Bayesian is to estimate the parameters of a model conditional on observed data ($P(\theta|D)$, known as *posterior distribution*) given the likelihood ($L(\theta|D) = P(D|\theta)$) and *prior distributions* of the parameters ($P(\theta)$), according to Bayes rule:

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}$$

Details on Bayes rule are given in section 4.3 and 6.2.2 of the book. Note that the only unknown in the right hand side of Bayes rule is $P(D)$. However, we know that $P(D) = \int P(D|\theta)P(\theta)d\theta$. Therefore, in order to calculate the posterior distribution, we need to calculate this integral. Any integration method would work but, in practice, the most popular ones are Markov Chain Monte Carlo (MCMC) algorithms due to their higher performance. These algorithms will provide a random sample from the posterior distribution given a formulation of the problem as:

$$\log(P(\theta|D)) \propto \log(P(D|\theta)) + \log(P(\theta)) = \mathcal{L} + \log(P(\theta))$$

Where $\mathcal{L}$ is the positive log-likelihood. The Bayesian algorithms work with logarithms for the same reason as in maximum likelihood estimation.

The first step is to build a function that calculates the log-posterior for every parameter value. For the negative binomial example we can use the following:

```
LPfun1 = function(p,dat=x) {
  p = exp(p)
  mu=p[1]
```

```
  lp_mu = dnorm(mu, 0, 1.9, log = TRUE)
  k=p[2]
  lp_k = dnorm(k, 0, 1.9, log = TRUE)
  LL = sum(dnbinom(dat,mu=mu,size=k,log=TRUE))
  LL + lp_mu + lp_k
}
```

The main difference with `NLLfun1` is that it now calculates the log-prior probabilities of `mu` and `k` (`lp_mu` and `lp_k`, respectively) and that it returns the sum of log-likelihood + log-priors. In this case, I am assuming that we have prior knowledge indicating that `mu` and `k` will be lower than 5 with 99% probability. There are three more subtle issues

- I added `p = exp(p)` at the beginning which means that the function will actually take `log(p)` as input. The reason is because the parameters of the negative binomial must be positive (this did not hurt in the previous exercises, but it can have a bigger impact in Bayesian problems).
- `LPfun1` does not actually calculate the log-posterior but a quantity proportional to it. The difference is $P(D)$, but that will be calculated (implictly) by the MCMC algorithm.
- I use a Normal distribution for priors, despite the fact that `mu` and `k` cannot be negative. There are called *improper* prior distributions because their areas are not 1. But that is not a problem as we can always multiply an improper prior by a constant (to make it proper) and merge that constant with $P(D)$.

Sampling from posterior: Metropolis-Hastings algorithm

Below is a simple version of the Metropolis-Hastings algorithm (section 7.3.1 of the book), with a multivariate Normal proposal distribution (**you need to install package `mvtnorm` first!**):

```
library(mvtnorm)
MH = function(model, init, sigma = diag(init/10), niter = 3e4, burn = 0.5,
                       seed = 1134, ...) {
  # To make results reproducible you should set a seed (change among chains!!!)
  set.seed(seed)
  # Preallocate chain of values
  chain = matrix(NA, ncol = length(init), nrow = niter)
  # Chain starts at init
  current = init
  lp_current = model(current, ...)
  # Iterate niter times and update chain
  for(iter in 1:niter) {
    # Generate a candidate value from multivariate Normal distribution
    candidate = rmvnorm(1, mean = current, sigma = sigma)
    # Calculate probability of acceptance (proposal distribution is symmetric)
    lp_candidate = model(candidate, ...)
    paccept = min(1, exp(lp_candidate - lp_current))
    # Accept the candidate...or not!
    # If accept, update the current and lp_current values
    accept = runif(1) < paccept
    if(accept) {
      chain[iter,] = candidate
      lp_current = lp_candidate
      current = chain[iter,]
    } else {
      chain[iter,] = current
    }
```

```
  }
  # Calculate acceptance probability after burn-in and the length of burn-in
  nburn = floor(niter*burn)
  acceptance = 1 - mean(duplicated(chain[-(1:nburn),]))
  # Package the results
  list(burnin = chain[1:nburn,], sample = chain[-(1:nburn),],
       acceptance = acceptance, nburn = nburn)
}
```

The inputs of `MH` are:

- `model`: Function that calculates the non-normalized log-posterior (i.e. `LPfun1`).
- `init`: Initial values for the parameters. The closer to the "true" values the faster the MCMC algorithm will converge to the posterior distribution.
- `sigma`: Variance-covariance matrix used to calculate jumps in parameter space.
- `niter`: Number of iterations the algorithm will run for.
- `burn`: Fraction of iterations that will be used as burn-in (check section 7.3.2). These iterations will not be used for analysis but are required for convergence of the MCMC algorithm.
- `seed`: Seed for pseudorandom number generator that allows reproducing results.

The algorithm keeps track of all the parameter values it visits and stores them in `chain`. Each iteration, it proposes a new candidate value (`candidate`) from a multivariate Normal distribution centered at the current value. The probability of accepting the candidate is equal to the difference in log posterior densities (`paccept`, see Equation 7.3.2 in the book, taking into account that the proposal distribution is symmetric). If the candidate value is accepted, then it is added to the `chain` and becomes the new `current` value (i.e. the algorithm "moves" to that location). After the run is finished, the values in `chain` are split according to burn-in and after burn-in. The variable `acceptance` calculates the fraction of jumps that were accepted. The higher this number, the more efficient the algorithm is in exploring the posterior distribution.

So let's tackle the negative binomial problem. First, let's regenerate the data just to be sure...

```
set.seed(1001)
mu.true=1
k.true=0.4
x = rnbinom(50,mu=mu.true,size=k.true)
```

Now we can run `MH` with some values. I want to make the point that the proposal distributions matters for an efficient MCMC algorithm. So let's start with a variance-covariance matrix is not reasonable:

```
sigma = diag(c(10,10))
```

Now we can run `MH`:

```
init = log(c(1,1))
bay1 = MH(LPfun1, init, sigma, burn = 0.3, dat = x)
```

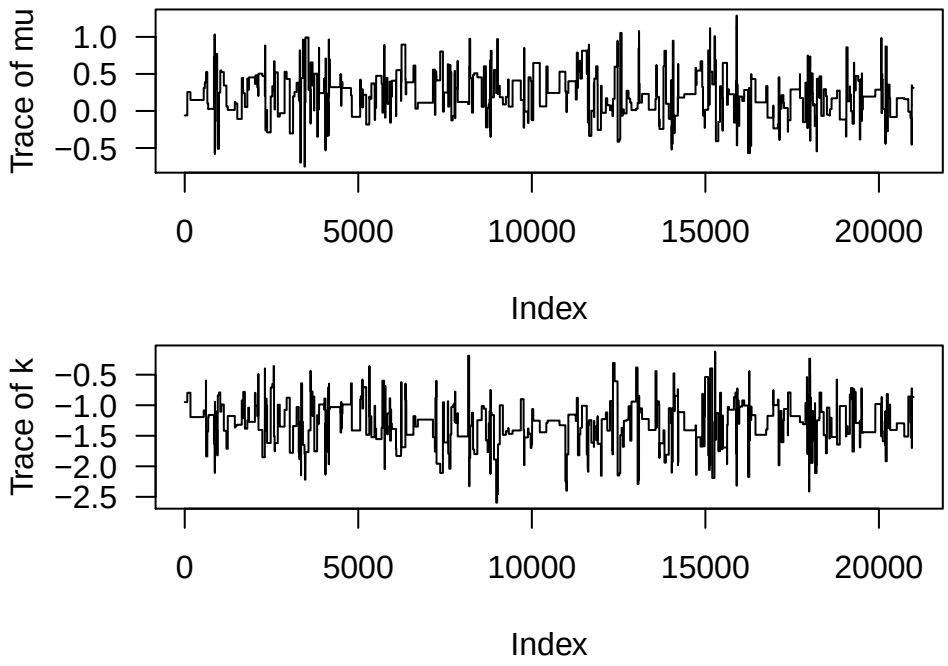
The first parameter you want to check is the acceptance probability:

```
bay1$acceptance
```

```
## [1] 0.02
```

This terrible! 98% of the candidate were rejected so it would take really long to get a representative sample from the posterior. The next step is usually to take a look at the traces of the values sampled by the MCMC:

```
par(mfrow = c(2,1), mar = c(4,4,0.5,0.5), las = 1)
plot(bay1$sample[,1], t = "l", ylab = "Trace of mu")
plot(bay1$sample[,2], t = "l", ylab = "Trace of k")
```



The low probability of acceptance means that the traces look like "squiggly lines", getting stuck at different values for hundres of iterations. This slows down the effective sampling and can introduce biases in the estimates (unless the algorithm runs for very, very long).

Modern MCMC algorithms will automatically *tune* the proposal distribution or even used alternatives methods to calculate candidate values that are more robust. However a poor man's tuned MCMC may suffice for this introduction and it works as follows:

1. Calculate the value that maximizes the posterior distribution (*a.k.a* MAP estimate).
2. Estimate the variance-covariance matrix of the posterior distribution using the Hessian matrix.
3. Run `MH` using the above as values `init` and `sigma`, respectively.

The reason why this works is because points 1 and 2 will often give a good first approximation of the posterior distribution, especially for large data. This means that `MH` will be sampling from a distribution similar to the target distribution. This approach can be implemented as:

```
mapfit = optim(fn = LPfun1, par = log(c(1,1)),
               hessian = TRUE, method = "BFGS",
               control = list(fnscale = -1), dat = x)
sigma = solve(-mapfit$hessian)
init = mapfit$par
bay2 = MH(LPfun1, init, sigma, burn = 0.3)
```

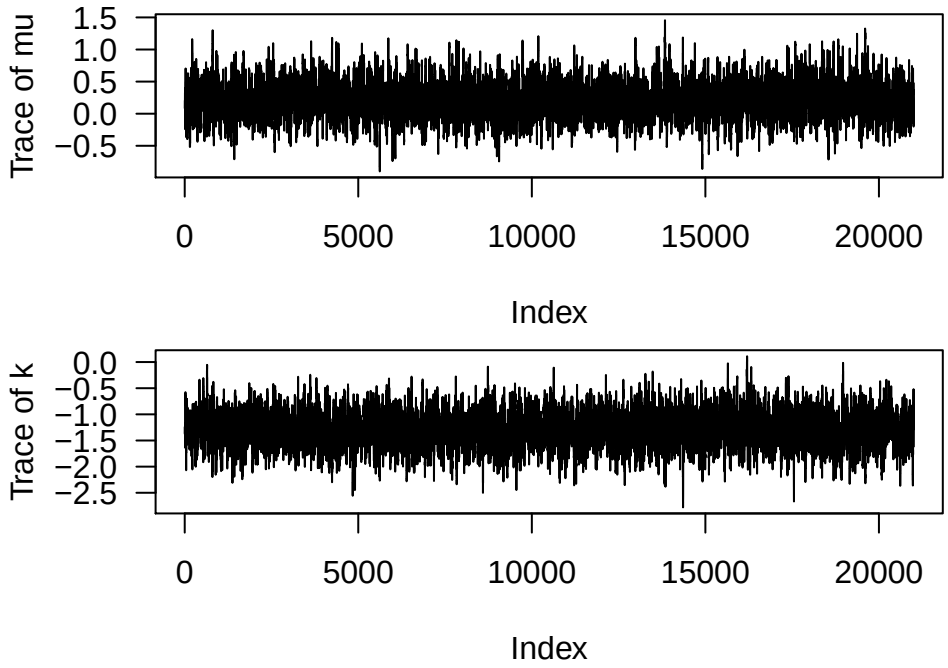
Notice that I used `control = list(fnscale = -1)` because I want to maximize the posterior probability, not minimize it. The new run has a higher acceptance probability:

```
bay2$acceptance
```

```
## [1] 0.5605238
```

Now this is the good. 56% of the time the candidates will be accepted, ensuring that the chain samples efficiently from the posterior. The traces will approach white noise (they look like "fuzzy catterpillars"):

```
par(mfrow = c(2,1), mar = c(4,4,0.5,0.5), las = 1)
plot(bay2$sample[,1], t = "l", ylab = "Trace of mu")
plot(bay2$sample[,2], t = "l", ylab = "Trace of k")
```

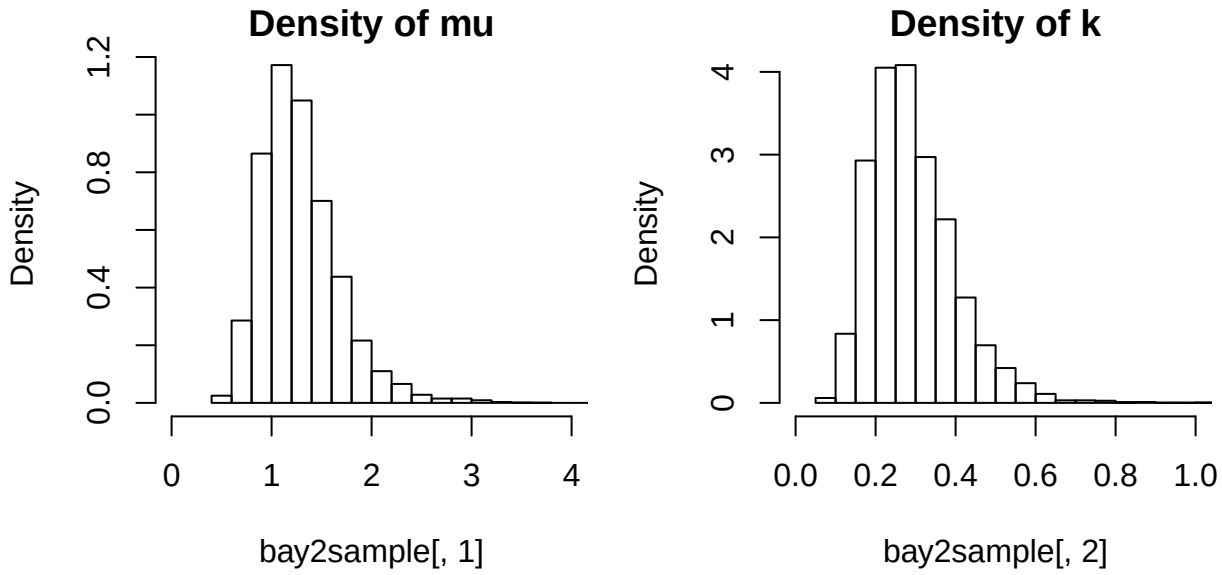


At this point you would normally calculate more diagnostics to build up more confidence on the results of the MCMC chains, but we will keep it simple in this introduction. The object `bay2$sample` contains a random sample from the posterior from which you can calculate several properties. First, remember that we took the logarithm of the parameters to avoid negative values, so we need to undo the transformation:

```
bay2sample = exp(bay2$sample)
```

We can visualize the estimates for each parameter using `density` (more common) or `hist` (easier to interpret):

```
par(mfrow = c(1,2), mar = c(4,4,1.5,1))
hist(bay2sample[,1], main = "Density of mu", freq = F, xlim = c(0,4))
hist(bay2sample[,2], main = "Density of k", freq = F, xlim = c(0,1))
```



One striking feature is that the distributions are not symmetric, they have a longer tail to the right. This is typical of positive parameters that are close to 0. A consequence of this is that the mean, median and mode of the distributions will differ (though in in this case not so much). Let's compare all the estimates we have so far:

```
map = exp(mapfit$par)
meanp = colMeans(bay2sample)
medianp = c(median(bay2sample[,1]), median(bay2sample[,2]))
cbind(map, meanp, medianp,
      mom = c(mu.mom, k.mom),
      mle = O1$par,
      true = c(mu.true, k.true))
```

```
##          map     meanp   medianp       mom       mle true
## mu 1.2170554 1.2866368 1.2245424 1.2600000 1.2602356  1.0
## k  0.2875031 0.2920738 0.2743753 0.3778531 0.2884793  0.4
```

For this model, data and priors, all estimates are quite similar to each other and to the estimate from maximum likelihood. The reason is because there is sufficient data (50 points for 2 parameters is quite some data...) such that the priors have a negligible effect.

The 95% credible intervals (analogous to 95% confidence intervals) can be calculated with the `quantile` function applied directly to the sample from the posterior:

```
t(apply(bay2sample, 2, quantile, probs = c(0.025, 0.975)))
```

```
##           2.5%    97.5%
## [1,] 0.7097222 2.231552
## [2,] 0.1361927 0.542852
```

The normal approximation could also be used (just like in maximum likelihood) but in this case these estimates are quite bad:

```
se = sqrt(diag(sigma))
cbind(map - 1.96*se, map + 1.96*se)
```

```
##            [,1]      [,2]
## [1,]  0.6685165 1.7655943
## [2,] -0.3993552 0.9743614
```

The reason they are so bad is that they miss the asymmetry completely, especially for `k`, where the predict negative values! This is the reason why we always need to run the MCMC algorithm and report probability intervals rather than standard errors.

Bayesian and covariates

In this section you practice Bayesian parameter estimation on the same dataset and model as the one you used in the section *Maximum likelihood and covariates*:

1. Think about what would be reasonable priors for the parameters. Extra information on the datasets is given below. Use this information to define some reasonable priors, considering the scale of measurements and what is known about the type of systems/populations being measured.
2. Use the likelihood function from previous exercise on this dataset, as well as the priors, to estimate the parameters of the model. Follow the approach shown for the negative binomial example (i.e. optimization followed by Metropolis-Hastings).
3. Calculate mean of posterior and 95% credible intervals. How much to they differ from the maximum likelihood estimates you obtained before?.
4. Repeat points 2 and 3 but making the priors wider to check how much they affect the posterior.

Hints for choosing deterministic functions and stochastic functions

1. Deterministic functions

   dataset 1 light response curve. There are a number of options of functions to choose from, depending on the level of sophistication: $\frac{ax}{(b+x)}$, $a(1-e^{(-bx)})$, $\frac{1}{2\theta}(\alpha I + p_{max} - \sqrt{(\alpha I + p_{max})^2 - 4\theta I p_{max}})$ see page 98. A parameter `d` can be added in all cases to shift the curve up or down. The y represents net photosynthesis $\mu molCO_2/m^2 s$

   dataset 2 The dataset describes a functional responses. Bolker mentions four of those $min(ax, s)$ $\frac{ax}{(b+x)}$, $\frac{ax^2}{(b^2+x^2)}$, $\frac{ax^2}{(b+cx+x^2)}$ The y is measured in grams prey eaten per unit time.

   dataset 3 Allometric relationships generally have the form ax^b. The y represent the total number of cones produced.

   dataset 4 This could be logistic growth $n(t) = \frac{K}{1+(\frac{K}{n_0})e^{-rt}}$ or the gompertz function $f(x) = e^{-ae^{-bx}}$. The y represent the population size (numbers).

   dataset 5 What about a negative exponential? $ae{-}bx$ or a power function ax^b. The y represent a number per unit area.

   dataset 6 Species reponse curves are curves that describe the probability of presence as a function of some factor. A good candidate good be a unimodel response curve. You could take the equation of the normal distribution without the scaling constant: e.g. $ae^{\frac{-(x-\mu)^2}{2\sigma^2}}$. The y represent presence or absence of the species (no units).

2. Stochastic functions/Probability distributions

   dataset 1 y represents real numbers and both positive and negative numbers occur. This implies that we should choose a continuous probability distribution. In addition, the numbers seems unbound. Within the family of continuous probability distributions, the normal seems a good candidate distribution because this one runs from -inf to +inf. In contrast the Gamma and the Lognormal only can take positive numbers, so these distributions cannot handle the negative numbers. In addition, the beta distribution is not a good candidate because it runs from 0-1.

   dataset 2 y represents real numbers and only positive numbers occur. The data represents a functional response (intake rate of the predator), and it is likely that you can only measure positive numbers (number of prey items per unit of time). This implies that we should choose a continuous probability distribution. Within the family of continuous probability distributions, the Gamma and the Lognormal could be taken as candidate distributions because they can only take positive numbers (beware that the Gamma cannot take 0). However, you could try to use a normal as well.

   dataset 3 y seems represents counts (this is the cone dataset that is introduced in ch. 6.). Given that it contains counts we can pick a distribution from the family of discrete distributions. The Poisson and the Negative Binomial could be good candidates to describe this type of data.

   dataset 4 y represents population size over time. From looking at the data, they seems to represent counts. Given that it contains counts we can pick a distribution from the family of discrete distributions. The Poisson and the Negative Binomial could be good candidates to describe this type of data.

   dataset 5 No information is given on y. The data clearly seems to represent counts. Thus the same reasoning applies here as to the two previous datasets.

   dataset 6 The data (y) represents species occurences (presence/absence). The binomial model would be a good model to predict the probability of presence.