Function to estimate phenology as a multi-modal Gaussian curve

Gaussian curve

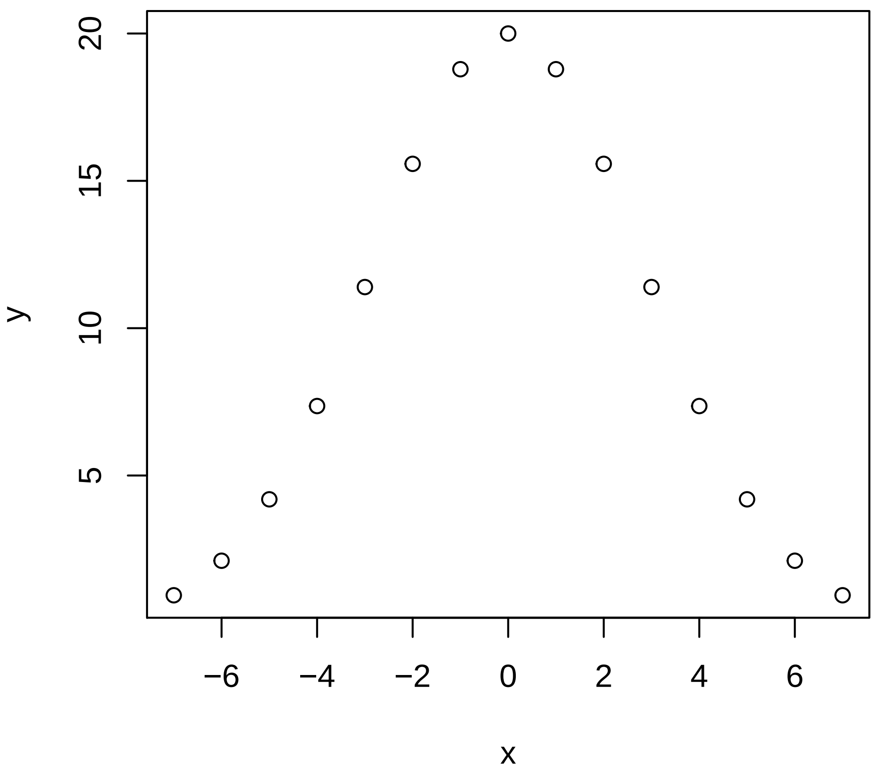
Let's get familiar with the shape and parameters of a Gaussian curve with one peak:

$$f(x) = a_1 \exp\left(-\frac{(x - a_2)^2}{2a_3^2}\right)$$

The parameter a_1 determines the maximum height of the curve. The parameter a_2 determines the location of the peak. The parameter a_3 determines how wide (larger value of a_3) or skinny (smaller a_3) the curve is.

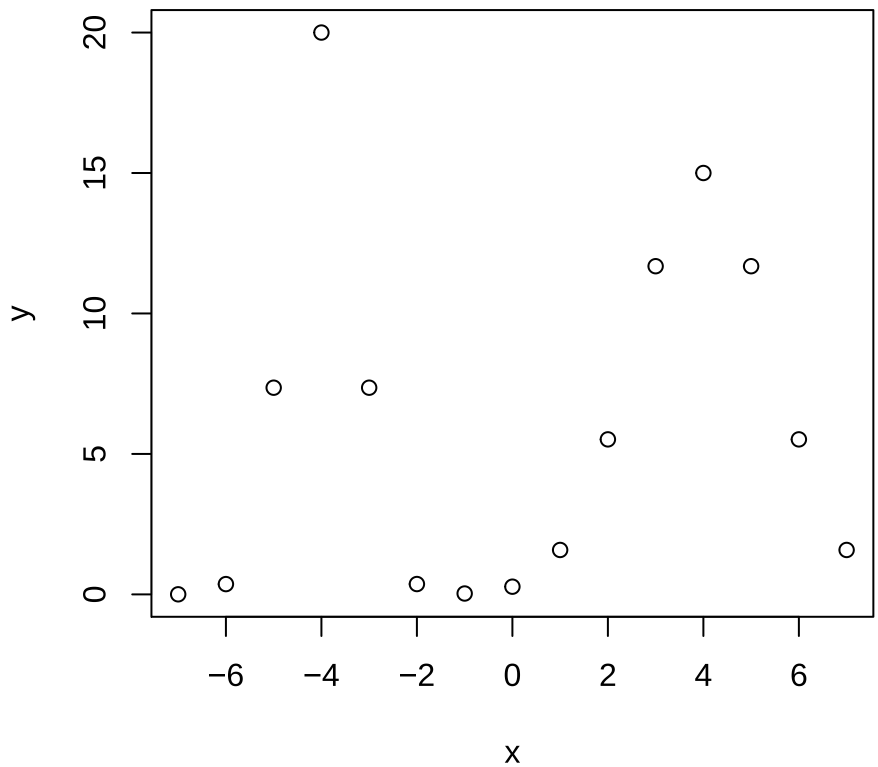
Let's make a funciton for this curve and plot it

```
gaus <- function(x, a1, a2, a3) {
    a1 * exp(-(x - a2)^2 / (2 * a3)^2)
}

x <- -7:7
y <- gaus(x, 20, 0, 2)
plot(x, y)
```



Now let's make a bi-modal Gaussian curve. This is simply a funciton of two Gaussian curves added together, each with its own set of parameters, say a_1, a_2, a_3 for the first curve, and b_1, b_2, b_3 for the second

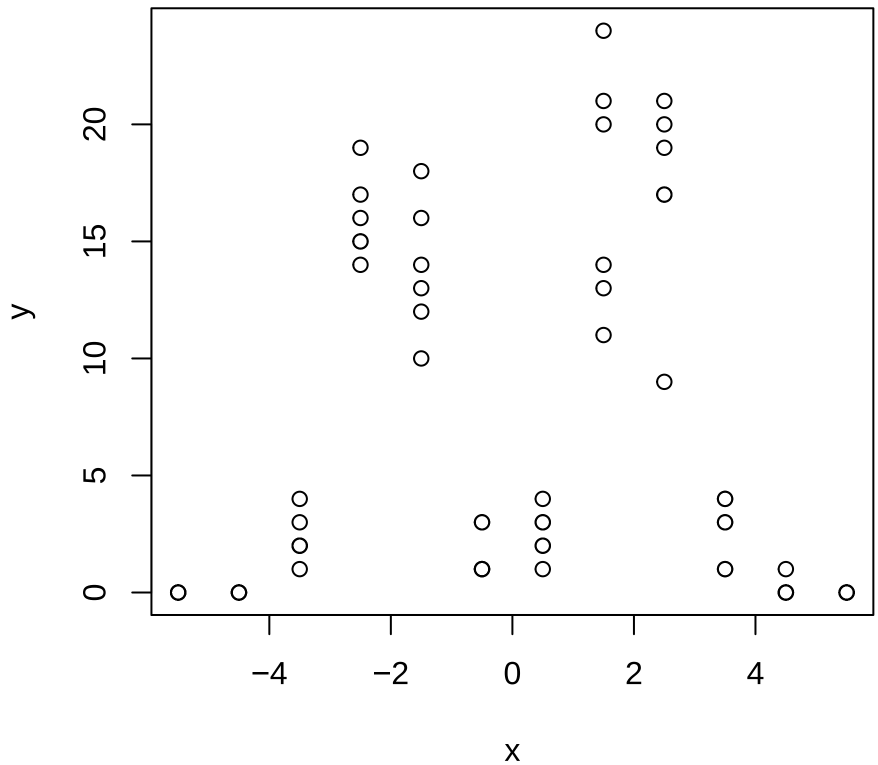
```r
gaus2 <- function(x, a1, a2, a3, b1, b2, b3) {
    a1 * exp(-(x - a2)^2 / (2 * a3)^2) + b1 * exp(-(x - b2)^2 / (2 * b3)^2)
}

x <- -7:7
y <- gaus2(x, 20, -4, 0.5, 15, 4, 1)
plot(x, y)
```



Bi-modal Gaussian data with noise

Our data will not be smooth like that, instead there will be noise. We can make a first attempt of modeling that noise with a Poisson distribution. This means that for every `x` value, the `y` response will actual be a random number drawn from a Poisson distribution whose mean value comes from the bi-model Gaussian curve. We can simulate data following that idea with an approach like this:

```r
# create realistic calendar sampling times, centered on mid-June
x <- rep((1:12) - 6.5, each = 6)

# for convenience, make a vector of our parameters a1, a2, a3, b1, b2, b3
p <- c(20, -2, 0.5, 20, 2, 0.5)

# now make the mean values for the y variable
ymean <- gaus2(x, p[1], p[2], p[3], p[4], p[5], p[6])

# finally, simulate real data
y <- rpois(length(ymean), ymean)

plot(x, y)
```



Fitting the curve to real data with noise

To fit the bi-model curve to these kind of real data, we need to define a log likelihood function and maximize it. We will actually make a *negative* log likelihood function and *minimize* it, but those tasks are equivalent, computers just prefer to minimize things. We'll be using the function `optim` to do that minimization (or *optimization*).

The model we've been building toward assumes there is a bi-model Gaussian curve with a potential early season peak and late season peak. The resulting data are assumed to come from a Poisson distribution. Taking that info together, we can make a negative log likelihood function that looks something like this:

```
gaus2LogLik <- function(p, x, y) {
    a1 <- p[1]
    a2 <- p[2]
    a3 <- p[3]
    b1 <- p[4]
    b2 <- p[5]
    b3 <- p[6]

    mu <- gaus2(x, a1, a2, a3, b1, b2, b3)

    return(-sum(dpois(y, mu, log = TRUE)))
}
```

A couple things to note about this function: we pass it three arguments: `p` is one vector containing all the parameter values (`a1`, `a2`, `a3`, `b1`, `b2`, `b3`) because that's how `optim` wants it to be; the vector `x` of dates samples were taken; and the vector `y` with the counts for one species.

In this function first we split up our vector `p` into the necessary parameters `a1`, `a2`, `a3`, `b1`, `b2`, `b3`, we then create a vector of mean values (`mu`) and then pass that to the Poisson distribution function `dpois`. What we return is the negative sum of the log values of the Poisson distribution function.

That's it! That's the negative log likelihood function.

But before using it, we need to do a few things. The optimization will have a hard time with the fact that in real life all the parameters `a1`, `a2`, `a3`, `b1`, `b2`, `b3` need to be positive, but the optimizer will naively try negative values. We can force it to only work in the positive domain by using the `exp` function like this:

```
gaus2LogLik <- function(p, x, y) {
    p <- exp(p)

    a1 <- p[1]
    a2 <- p[2]
    a3 <- p[3]
    b1 <- p[4]
    b2 <- p[5]
    b3 <- p[6]

    mu <- gaus2(x, a1, a2, a3, b1, b2, b3)

    return(-sum(dpois(y, mu, log = TRUE)))
}
```

That's very close to what we need, but the optimizer will have a hard time with one last thing: right now, it doesn't know that we want it to force an early season peak and a late season peak. But we can make it do that. That's why we centered the `x` values at mid June, i.e. `6.5` (this could be modified to whatever makes sense, e.g. `6.75`). Assuming we've done that kind of centering, we can force the first peak to be before "mid season" by forcing it to always be negative, like this:

```
gaus2LogLik <- function(p, x, y) {
    p <- exp(p)

    a1 <- p[1]
    a2 <- -p[2]
    a3 <- p[3]
    b1 <- p[4]
    b2 <- p[5]
    b3 <- p[6]

    mu <- gaus2(x, a1, a2, a3, b1, b2, b3)

    return(-sum(dpois(y, mu, log = TRUE)))
}
```

Now we've got it. Let's test out this function with our simulated data from before:

```
# remember, we saved that vector of parameters
p
```

```
## [1] 20.0 -2.0  0.5 20.0  2.0  0.5
```

```
# but our new log likelihood function has that funny business with exponentials and
# negatives, so let's make a new p to use with the log likelihood (ll) function
pll <- p
pll[2] <- -pll[2]
pll <- log(pll)
pll
```

```
## [1]  2.9957323  0.6931472 -0.6931472  2.9957323  0.6931472 -0.6931472
```

```r
# and remember the data vectors `x` and `y` exist (and `x` is centered at 6.5)
x
```

```
##  [1] -5.5 -5.5 -5.5 -5.5 -5.5 -5.5 -4.5 -4.5 -4.5 -4.5 -4.5 -4.5 -3.5 -3.5 -3.5
## [16] -3.5 -3.5 -3.5 -2.5 -2.5 -2.5 -2.5 -2.5 -2.5 -1.5 -1.5 -1.5 -1.5 -1.5 -1.5
## [31] -0.5 -0.5 -0.5 -0.5 -0.5 -0.5  0.5  0.5  0.5  0.5  0.5  0.5  1.5  1.5  1.5
## [46]  1.5  1.5  1.5  2.5  2.5  2.5  2.5  2.5  2.5  3.5  3.5  3.5  3.5  3.5  3.5
## [61]  4.5  4.5  4.5  4.5  4.5  4.5  5.5  5.5  5.5  5.5  5.5  5.5
```

```r
y
```

```
##  [1]  0  0  0  0  0  0  0  0  0  0  0  0  2  3  2  2  4  1 15 19 16 15 17 14 10
## [26] 12 16 18 13 14  1  3  3  1  1  1  4  1  3  2  2  3 24 20 21 11 14 13 21 19
## [51] 17 17  9 20  3  1  3  4  1  4  0  0  1  0  0  0  0  0  0  0  0  0
```

```r
# so what is the log likelihood at for the parameter values p
gaus2LogLik(pll, x, y)
```

```
## [1] 107.7651
```

```r
# compare that to something with different parameter values
pllWrong <- pll
pllWrong[c(1, 3, 5)] <- 1
pllWrong
```

```
## [1]  1.0000000  0.6931472  1.0000000  2.9957323  1.0000000 -0.6931472
```

```r
gaus2LogLik(pllWrong, x, y)
```

```
## [1] 368.1458
```

The vector of parameters `pllWrong` is "wrong" in the sense that those were not the parameter values used to simulate the data `y`. So if everything is working right, the negative log likelihood should be **bigger** for the wrong values, and it is.

Now let's use the function `optim` to minimize the negative log likelihood function for the simulated data. This is exactly what we'll do for real data too! One thing to note, we have to pass `optim` a vector of initial guesses for the parameter values, we'll start by using `pll`.

```r
optim(pll, gaus2LogLik, x = x, y = y, method = 'BFGS', hessian = TRUE)
```

```
## $par
## [1]  2.9670793  0.7241617 -0.7124154  3.0672742  0.6997575 -0.6510332
##
## $value
## [1] 104.8515
##
## $counts
## function gradient
##       51       11
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##             [,1]        [,2]       [,3]        [,4]        [,5]       [,6]
```

```
## [1,] 202.2058087    4.064750 199.649267   0.5084198   -4.293692   5.038856
## [2,]   4.0647503 1766.173549  28.687353  -3.9125246   31.551028 -35.771255
## [3,] 199.6492667   28.687353 568.900687   3.5987222  -27.432592  29.727795
## [4,]   0.5084198   -3.912525   3.598722 237.7771123    4.304577 233.468240
## [5,]  -4.2936917   31.551028 -27.432592   4.3045772 1739.726193  44.945175
## [6,]   5.0388562  -35.771255  29.727795 233.4682400   44.945175 656.237329
```

Note really quick how we call this function: first we pass the initial guess `pll`, then the function we optimizing `gaus2LogLik`, and then we also have to tell `optim` the values of the `x` and `y` parameters. Finally, we tell optim to use the `BFGS` optimization method, and we tell it to keep the `hessian` (more on that later).

The function `optim` returns a lot of information, all in one list object. The the important things to note are: the element called `$par` gives the maximum likelihood estimates, `$value` gives the optimal negative log likelihood, and `$convergence` tells us if the optimization was successful: a value of `0` is success, any other value is not, you can learn more about the other values in the help doc `?optim`. Excitingly, the model estimates in `$par` are very close to the parameter values (`pll`) that we used to simulate the data. That tells us our estimation procedure is working. We should also check that if we supply `optim` with a different initial guess, that it still converges on this correct answer. So let's try using `pllWrong` as our initial guess:

```r
optim(pllWrong, gaus2LogLik, x = x, y = y, method = 'BFGS', hessian = TRUE)
```

```
## $par
## [1]  2.9670795  0.7241618 -0.7124152  3.0672746  0.6997574 -0.6510325
##
## $value
## [1] 104.8515
##
## $counts
## function gradient
##       82       23
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##              [,1]        [,2]       [,3]        [,4]        [,5]       [,6]
## [1,] 202.2058697    4.064769 199.649317   0.5084229   -4.293715   5.038887
## [2,]   4.0647692 1766.173203  28.687168  -3.9125435   31.551164 -35.771443
## [3,] 199.6493168   28.687168 568.900569   3.5987347  -27.432678  29.727925
## [4,]   0.5084229   -3.912544   3.598735 237.7773730    4.304600 233.468470
## [5,]  -4.2937149   31.551164 -27.432678   4.3045999 1739.722664  44.945972
## [6,]   5.0388870  -35.771443  29.727925 233.4684705   44.945972 656.236779
```

It gets the same answer so that's great!

Keeping in mind that these estimates are in log units, let's make a little convenience function to put them back into linear units:

```r
getParams <- function(p) {
    p <- exp(p)
    p[2] <- -p[2]

    return(p)
}
```

```r
mod <- optim(pllWrong, gaus2LogLik, x = x, y = y, method = 'BFGS', hessian = TRUE)
getParams(mod$par)
```

```
## [1] 19.4350758 -2.0630012  0.4904582 21.4832728  2.0132641  0.5215071
```

Lastly, a word on the Hessian. This is a matrix of partial derivatives of the likelihood function. It can be used to estimate the standard errors and confidence intervals of each parameter. We won't go into that now, but keep it in mind if we want to incorporate that later. I wrote a custom package a while back to do that kind of thing (the package is called *socorro*, named after the city, and because it's a package of helper functions). I'll put the code here for the sake of completeness. Before running this code, make sure you install the package *devtools*.

```r
library(devtools)
```

```
## Loading required package: usethis
```

```r
# check if socorro has already been installed
socorroCheck <- require(socorro)
```

```
## Loading required package: socorro
```

```r
# if it hasn't been installed, install it now
if(!socorroCheck) {
    install_github('ajrominger/socorro')
}

library(socorro)

# parameter confidence intervals
modCI <- wald(mod$par, mod$hessian, alpha = 0.05, marginal = TRUE)
modCI
```

```
##            2.5%      97.5%
## [1,]  2.7965152  3.1376437
## [2,]  0.6774560  0.7708676
## [3,] -0.8143293 -0.6105012
## [4,]  2.9095843  3.2249649
## [5,]  0.6526782  0.7468365
## [6,] -0.7462235 -0.5558414
```

Remember, these are in the funny units we use in the optimization. We can convert them to regular units, again using our convenience function `getParams`

```r
cbind(getParams(modCI[, 1]), getParams(modCI[, 2]))
```

```
##            [,1]       [,2]
## [1,] 16.3874407 23.0494912
## [2,] -1.9688626 -2.1616410
## [3,]  0.4429363  0.5430786
## [4,] 18.3491699 25.1526916
## [5,]  1.9206779  2.1103135
## [6,]  0.4741538  0.5735894
```