Function to estimate phenology as a multi-modal Gaussian curve

Gaussian curve

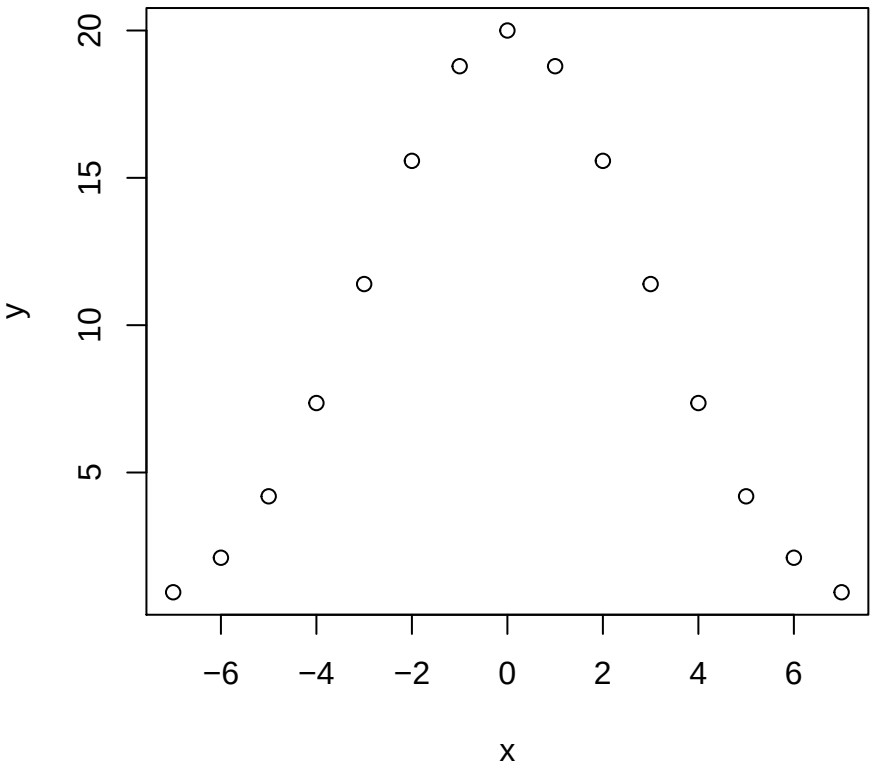
Let's get familiar with the shape and parameters of a Gaussian curve with one peak:

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

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.

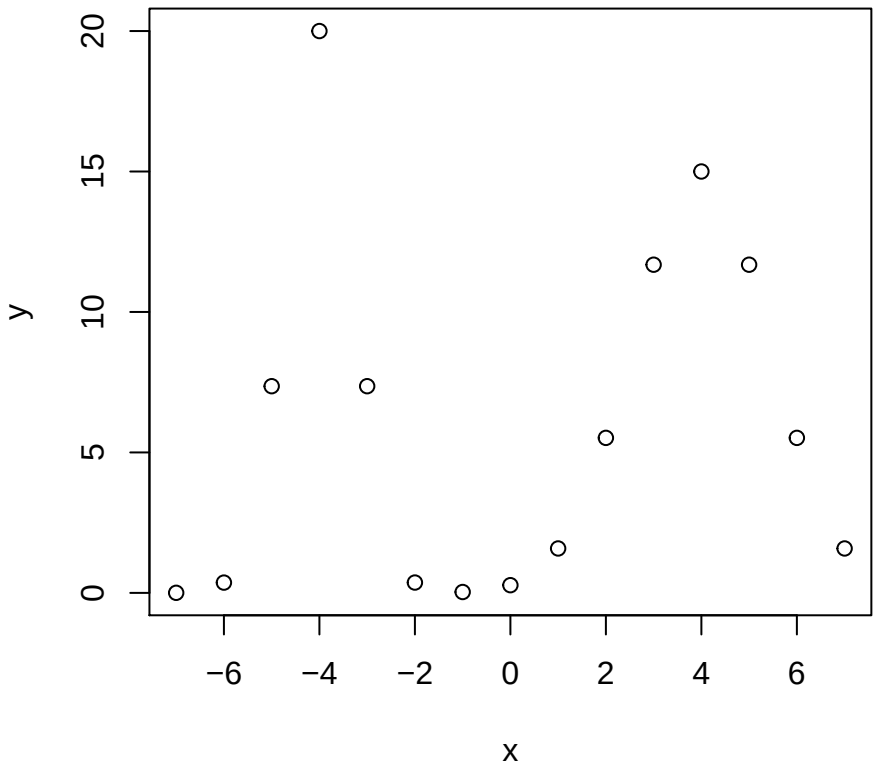
Let's make a funciton for this curve and plot it

```
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 it's 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

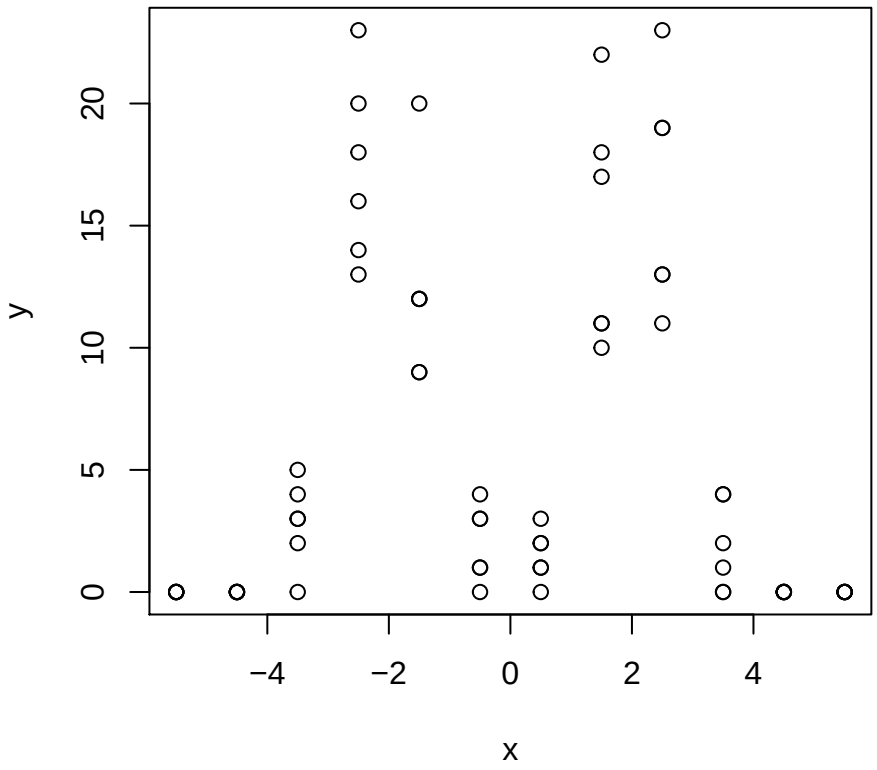
```r
gaus2 <- function(x, a1, a2, a3, b1, b2, b3) {
    a1 * exp(-(x - a2)^2 / (2 * a3)^2) + b1 * exp(-(x - b2)^2 / (2 * b3)^2)
}

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 will be usign the package *gnlm* to do this negative log likelihood minimization. Before proceeding, make sure that package is installed with `install.packages('gnlm')`.

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) {
    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 one argument, `p` because that's how the package *gnlm* wants it to be. We also assume that the objects `x` and `y` **already exist** in our working environment, which is why we can use them inside the function without passing them to the function as arguments. This is not great coding practice, but it's now the functions in *gnlm* were designed to work.

With those caveats in mind, what we're doing in this function is very simple: after splitting 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) {
    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) {
    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
```

```
# and remember the data vectors `x` and `y` exist
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
```

```
y
```

```
##  [1]  0  0  0  0  0  0  0  0  0  0  0  0  0  3  5  4  2  3 18 14 16 20 13 23 20
## [26]  9 12 12  9 12  3  1  3  4  1  0  3  2  1  2  0  1 10 11 18 22 11 17 13 19
## [51] 13 23 19 11  4  1  0  2  4  0  0  0  0  0  0  0  0  0  0  0  0  0
```

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

```
## [1] 113.4884
```

```
# 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
```

```
gaus2LogLik(pllWrong)
```

```
## [1] 372.8212
```

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 `gnlr` from package *gnlm* to optimize 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 `gnlr` a vector of initial guesses for the parameter values, we'll start by using `pll`.

```
library(gnlm)
```

```
## Loading required package: rmutil
```

```
##
## Attaching package: 'rmutil'
```

```
## The following object is masked from 'package:stats':
##
##     nobs
```

```
## The following objects are masked from 'package:base':
##
##     as.data.frame, units
```

```
gnlr(y, dist = gaus2LogLik, pmu = pll)
```

```
## Warning in nlm(fcn, p = p, hessian = TRUE, print.level = print.level, typsize =
## typsize, : NA/Inf replaced by maximum positive value
```

```
##
## Call:
```

```
## gnlr(y, dist = gaus2LogLik, pmu = pll)
##
## own distribution
##
## Response: y
##
## Log likelihood function:
## {
##     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)))
## }
##
## -Log likelihood    109.8607
## Degrees of freedom 66
## AIC                115.8607
## Iterations         9
##
## Model parameters:
##       estimate       se
## p[1]    2.9473  0.08592
## p[2]    0.7453  0.02398
## p[3]   -0.6715  0.05115
## p[4]    3.0315  0.08643
## p[5]    0.7110  0.02274
## p[6]   -0.7561  0.05167
##
## Correlations:
##           1         2        3         4         5        6
## 1  1.000000  0.012751 -0.58801 -0.006623 -0.010764  0.01686
## 2  0.012751  1.000000 -0.02794 -0.009746 -0.012434  0.02241
## 3 -0.588013 -0.027942  1.00000  0.016221  0.023901 -0.03956
## 4 -0.006623 -0.009746  0.01622  1.000000  0.009786 -0.59416
## 5 -0.010764 -0.012434  0.02390  0.009786  1.000000 -0.02245
## 6  0.016860  0.022410 -0.03956 -0.594165 -0.022454  1.00000
```

This returns a lot of information, but one important thing to look at is the info under `Model parameters`. This tells us the model parameter estimates, and excitingly, they're 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 `gnlr` with a different initial guess, that it still converges on this correct answer. So let's try using `pllWrong` as our initial guess:

```
gnlr(y, dist = gaus2LogLik, pmu = pllWrong)
```

```
##
## Call:
## gnlr(y, dist = gaus2LogLik, pmu = pllWrong)
##
## own distribution
```

```
##
## Response: y
##
## Log likelihood function:
## {
##     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)))
## }
##
## -Log likelihood    109.8607
## Degrees of freedom 66
## AIC                115.8607
## Iterations         18
##
## Model parameters:
##       estimate       se
## p[1]    2.9473  0.08593
## p[2]    0.7453  0.02398
## p[3]   -0.6715  0.05115
## p[4]    3.0315  0.08643
## p[5]    0.7110  0.02274
## p[6]   -0.7561  0.05167
##
## Correlations:
##           1         2        3         4         5        6
## 1  1.000000  0.012766 -0.58810 -0.006625 -0.010772  0.01687
## 2  0.012766  1.000000 -0.02796 -0.009745 -0.012428  0.02241
## 3 -0.588104 -0.027963  1.00000  0.016221  0.023906 -0.03957
## 4 -0.006625 -0.009745  0.01622  1.000000  0.009596 -0.59416
## 5 -0.010772 -0.012428  0.02391  0.009596  1.000000 -0.02213
## 6  0.016871  0.022408 -0.03957 -0.594163 -0.022135  1.00000
```

It gets the same answer under `Model parameters`, so that's great!