Curve fitting How−to

by W. Garrett Mitchener

This worksheet goes over traditional linear and non−linear least squares curve fitting and different ways to do it in *Mathematica*. It also goes over maximum likelihood curve fitting. Along the way, it shows different functions for finding maxima and minima of expressions.

Least squares and linear regression

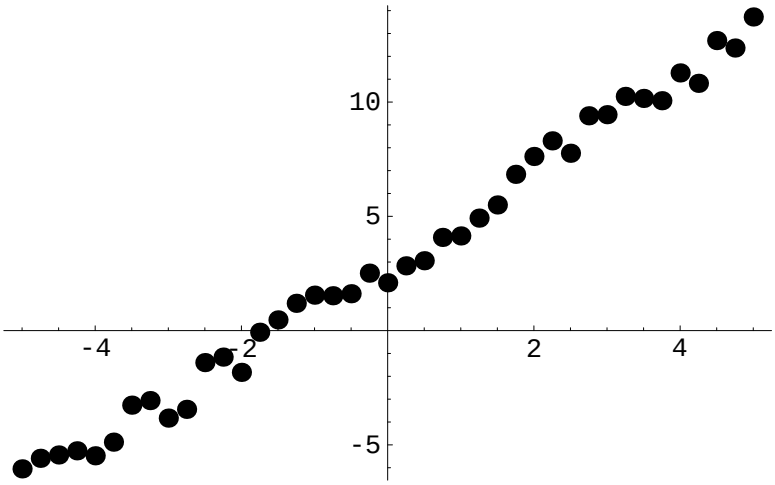
Let's say you have some data $\{(x_1, y_1), (x_2, y_2) ...\}$ and you want to fit a curve to it so that you can say $y = f(x) + \text{noise}$.

To illustrate, let's create some noisy data:

```
data = Table[{x, 2 x + 3 + Random[Real, {-1, 1}]},
  {x, -5, 5, 0.25}]
```

```
{{-5, -6.04965}, {-4.75, -5.58767}, {-4.5, -5.44105},
 {-4.25, -5.25967}, {-4., -5.47726}, {-3.75, -4.88052},
 {-3.5, -3.26153}, {-3.25, -3.062}, {-3., -3.82822},
 {-2.75, -3.44486}, {-2.5, -1.39979}, {-2.25, -1.15936},
 {-2., -1.82715}, {-1.75, -0.0717861}, {-1.5, 0.468823},
 {-1.25, 1.19398}, {-1., 1.5513}, {-0.75, 1.52568},
 {-0.5, 1.61352}, {-0.25, 2.51395}, {0., 2.09272},
 {0.25, 2.83461}, {0.5, 3.05718}, {0.75, 4.07795}, {1., 4.14236},
 {1.25, 4.92228}, {1.5, 5.49823}, {1.75, 6.83762}, {2., 7.61963},
 {2.25, 8.30281}, {2.5, 7.75976}, {2.75, 9.39962}, {3., 9.44785},
 {3.25, 10.2477}, {3.5, 10.1595}, {3.75, 10.059}, {4., 11.275},
 {4.25, 10.8195}, {4.5, 12.6907}, {4.75, 12.365}, {5., 13.7237}}
```

```
pointPlot = ListPlot[data, PlotStyle → PointSize[0.025]]
```



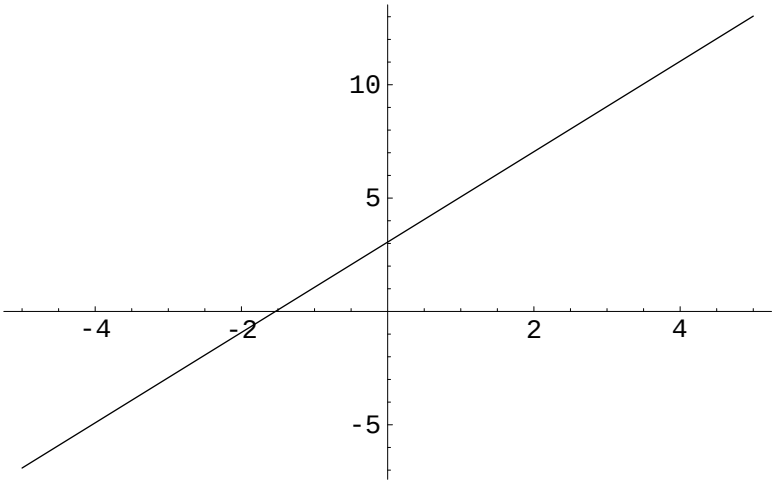

```
- Graphics -
```

In *Mathematica*, the `Fit` function takes a list of points, a list of expressions, and a list of independent variables, and determines which linear combination of the given expressions produces the best fit to the data. If you want to fit a line to the data, your list of expressions should consist of 1 and x, since a line is a linear combination of a constant and a multiple of x:

```
lineFit = Fit[data, {1, x}, x]
```

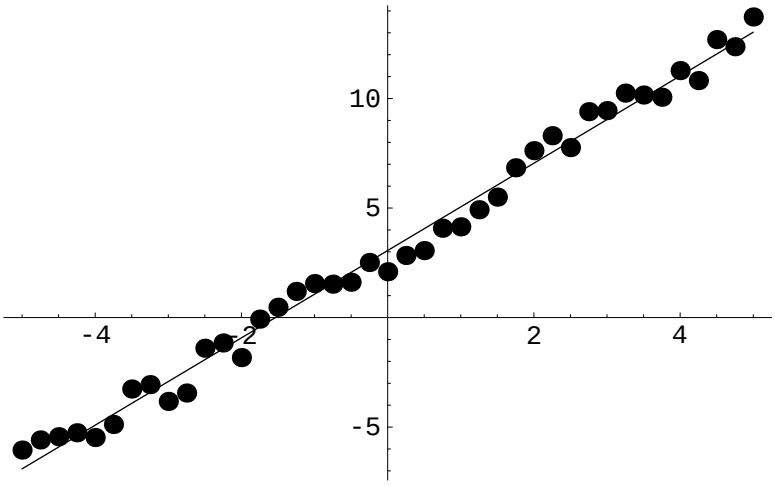
```
3.05974 + 1.9935 x
```

```
linePlot = Plot[lineFit, {x, -5, 5}]
```




```
- Graphics -
```

```
Show[pointPlot, linePlot]
```




```
- Graphics -
```

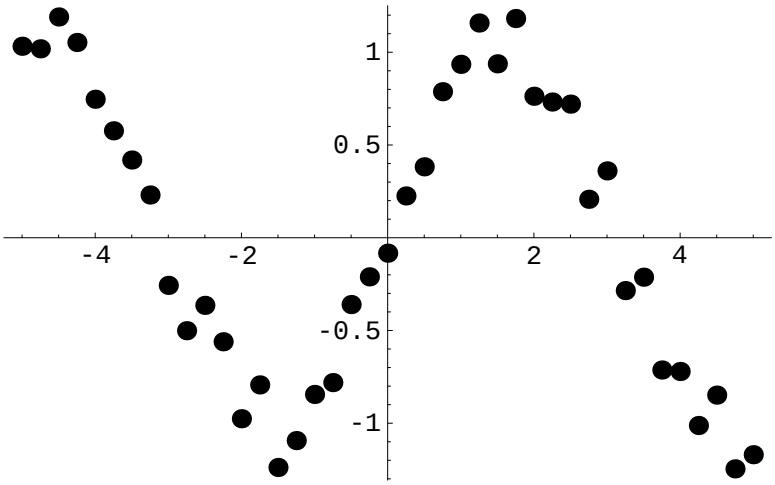
And you can see that we've more or less recovered the $3 + 2\,x$ that we used to create the data in the first place.

You can also do more interesting things:

```
data2 = Table[{x, Sin[x] + Random[Real, {-0.25, 0.25}]},
  {x, -5, 5, 0.25}]
```

```
{{-5, 1.03237}, {-4.75, 1.01859}, {-4.5, 1.19029}, {-4.25, 1.05274},
 {-4., 0.746595}, {-3.75, 0.576567}, {-3.5, 0.419058},
 {-3.25, 0.230351}, {-3., -0.256899}, {-2.75, -0.501214},
 {-2.5, -0.364602}, {-2.25, -0.560824}, {-2., -0.975778},
 {-1.75, -0.79348}, {-1.5, -1.23853}, {-1.25, -1.0937},
 {-1., -0.844869}, {-0.75, -0.781019}, {-0.5, -0.360202},
 {-0.25, -0.210867}, {0., -0.0832628}, {0.25, 0.225342},
 {0.5, 0.382401}, {0.75, 0.78725}, {1., 0.934763}, {1.25, 1.15762},
 {1.5, 0.937709}, {1.75, 1.18185}, {2., 0.762797}, {2.25, 0.731705},
 {2.5, 0.720411}, {2.75, 0.20737}, {3., 0.360398}, {3.25, -0.285011},
 {3.5, -0.212714}, {3.75, -0.713101}, {4., -0.721044},
 {4.25, -1.01231}, {4.5, -0.848428}, {4.75, -1.24612}, {5., -1.16977}}
```

```
pointPlot2 = ListPlot[data2, PlotStyle → PointSize[0.025]]
```



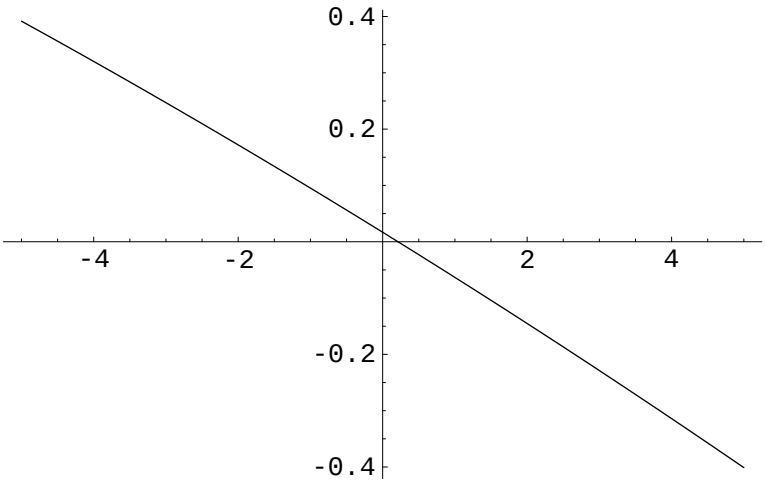

```
- Graphics -
```

For instance, this fits a second degree polynomial to the data:

```
poly2fit = Fit[data2, {1, x, x^2}, x]
```

$$0.0168822 - 0.0792803\,x - 0.000863356\,x^2$$

```
Plot[poly2fit, {x, -5, 5}]
```



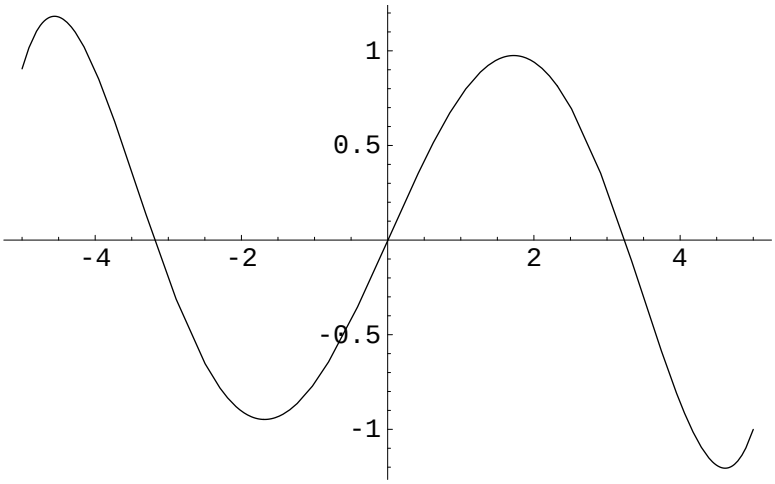

```
- Graphics -
```

Not a very good fit, is it? It doesn't have nearly enough bumps. Let's try something higher degree.

```
poly5fit = Fit[data2, {1, x, x^2, x^3, x^4, x^5}, x]
```

$$-0.00380876 + 0.870284\,x + 0.00704091\,x^2 - 0.11371\,x^3 - 0.000351999\,x^4 + 0.00285105\,x^5$$

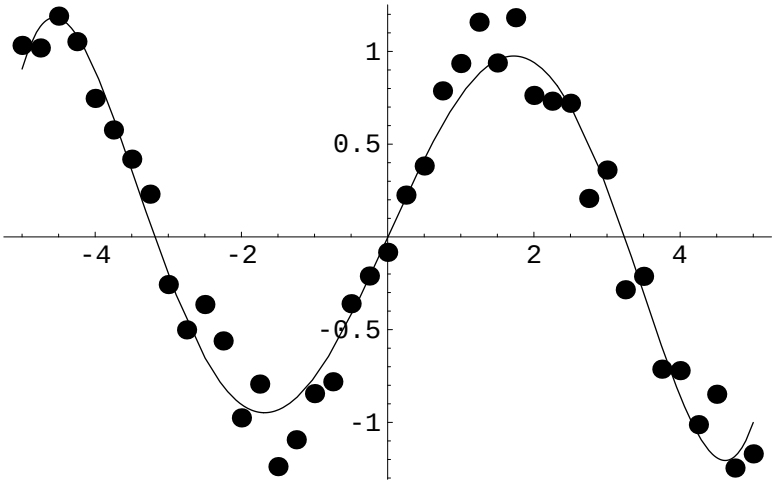
```
poly5plot = Plot[poly5fit, {x, -5, 5}]
```




```
- Graphics -
```

That looks a lot better. I knew to guess 5 for the degree because the data goes through four extrema.

```
Show[poly5plot, pointPlot2]
```




```
- Graphics -
```

But don't get carried away. If you give it too many degrees of freedom, it will start to fit the noise, as in this example:

```
powerTable = Table[x^n, {n, 0, 25}]
```

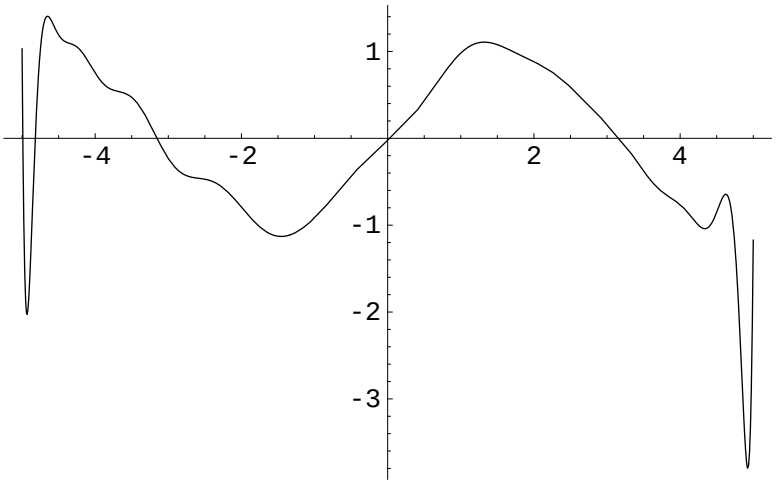
$$\{1, x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^{10}, x^{11}, x^{12}, x^{13}, x^{14}, x^{15}, x^{16}, x^{17}, x^{18}, x^{19}, x^{20}, x^{21}, x^{22}, x^{23}, x^{24}, x^{25}\}$$

```
poly25fit = Fit[data2, powerTable, x]
```

— *General::spell : Possible spelling error: new symbol name "poly25fit" is similar to existing symbols {poly2fit, poly5fit}.* More...

$$-0.0240747 + 0.736313\,x + 0.059761\,x^2 + 0.670674\,x^3 + 0.142823\,x^4 - 0.690569\,x^5 - 0.260176\,x^6 + 0.303735\,x^7 + 0.152971\,x^8 - 0.0858424\,x^9 - 0.0457667\,x^{10} + 0.0167881\,x^{11} + 0.00808788\,x^{12} - 0.00228625\,x^{13} - 0.000904123\,x^{14} + 0.00021515\,x^{15} + 0.0000657761\,x^{16} - 0.0000138214\,x^{17} - 3.1092\times10^{-6}\,x^{18} + 5.92366\times10^{-7}\,x^{19} + 9.21323\times10^{-8}\,x^{20} - 1.61591\times10^{-8}\,x^{21} - 1.55495\times10^{-9}\,x^{22} + 2.53384\times10^{-10}\,x^{23} + 1.1405\times10^{-11}\,x^{24} - 1.73697\times10^{-12}\,x^{25}$$

```
Plot[poly25fit, {x, -5, 5}]
```



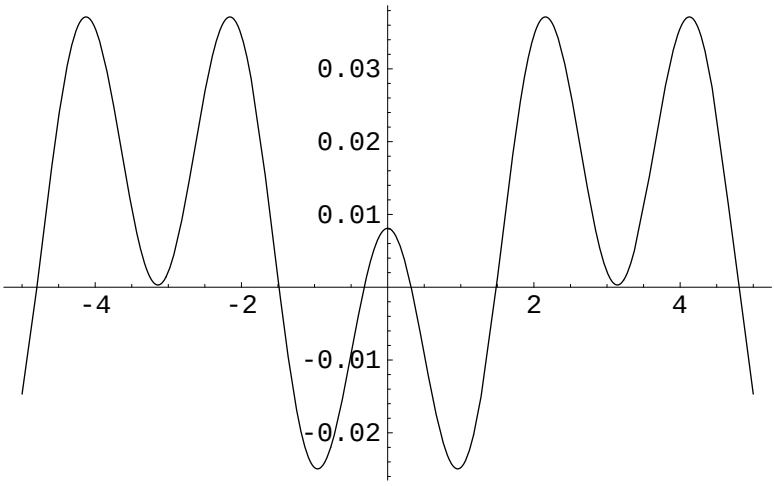

```
- Graphics -
```

You can also do more exotic things:

```
cos3fit = Fit[data2, {1, Cos[x], Cos[2 x], Cos[3 x]}, x]
```

$$0.00558514 - 0.017702\,\text{Cos}[x] - 0.00138422\,\text{Cos}[2x] + 0.0215951\,\text{Cos}[3x]$$

```
Plot[cos3fit, {x, -5, 5}]
```




```
- Graphics -
```

but that doesn't look right somehow. In fact, I get vastly different plots every time I run this worksheet, which indicates that the random noise added to the data is having a huge impact on the cosine fit, which isn't the case for the fifth degree polynomial. That indicates that these cosines are not a good way to fit this data. (Think about it: Why?) The first step to getting a good fit is to know what functions to include.

■ Details on how `Fit` works.

The way `Fit` works is called least squares, because it minimizes this:

$$\sum_{j=1}^{n} (f[x_j] - y_j)^2$$

In *Mathematica* notation

```
LeastSquaresError[data_, f_] :=
 Sum[(f[data[[j, 1]]] - data[[j, 2]])^2, {j, 1, Length[data]}]
```

Let's return to our linear data.

```
data
```

```
{{-5, -6.04965}, {-4.75, -5.58767}, {-4.5, -5.44105},
 {-4.25, -5.25967}, {-4., -5.47726}, {-3.75, -4.88052},
 {-3.5, -3.26153}, {-3.25, -3.062}, {-3., -3.82822},
 {-2.75, -3.44486}, {-2.5, -1.39979}, {-2.25, -1.15936},
 {-2., -1.82715}, {-1.75, -0.0717861}, {-1.5, 0.468823},
 {-1.25, 1.19398}, {-1., 1.5513}, {-0.75, 1.52568},
 {-0.5, 1.61352}, {-0.25, 2.51395}, {0., 2.09272},
 {0.25, 2.83461}, {0.5, 3.05718}, {0.75, 4.07795}, {1., 4.14236},
 {1.25, 4.92228}, {1.5, 5.49823}, {1.75, 6.83762}, {2., 7.61963},
 {2.25, 8.30281}, {2.5, 7.75976}, {2.75, 9.39962}, {3., 9.44785},
 {3.25, 10.2477}, {3.5, 10.1595}, {3.75, 10.059}, {4., 11.275},
 {4.25, 10.8195}, {4.5, 12.6907}, {4.75, 12.365}, {5., 13.7237}}
```

Here's how to define a general line function:

```
lineFunction[x_] = a x + b
```

```
b + a x
```

To fit this line to the data, we need to determine a and b such that the error is minimized.

Mathematica has several functions for finding the minimum of an expression. They all work a little differently. The `Minimize` function works algebraically:

```
minSolution = Minimize[LeastSquaresError[data, lineFunction], {a, b}]
```

```
{15.3329, {a → 1.9935, b → 3.05974}}
```

The result is a list {min, args} where min is the minimum value it found, and args is a rule table. Here's how to use a rule table. First, just so we're clear on what's going on, we'll unpack the list returned by `Minimize`.

```
{min, args} = minSolution
```

```
– General::spell1 : Possible spelling error: new
    symbol name "min" is similar to existing symbol "Min". More...
```

```
{15.3329, {a → 1.9935, b → 3.05974}}
```

That defined min to be the minimum value, and args to be the rule table giving the values of a and b:

```
args
```

```
{a → 1.9935, b → 3.05974}
```

This was the function we were trying to fit:

```
lineFunction[x]
```

```
b + a x
```

And we can connect the general function to the specific values of a and b by using the `/.` operator. (You can also use the `ReplaceAll` function. The `/.` is short−hand for `ReplaceAll`.)

```
lineFunction[x] /. args
```

```
3.05974 + 1.9935 x
```

```
ReplaceAll[lineFunction[x], args]
```

```
3.05974 + 1.9935 x
```

If you want to define a function representing the fit:

```
fittedFunction[x_] = lineFunction[x] /. args
```

```
3.05974 + 1.9935 x
```

```
fittedFunction[3.5]
```

```
10.037
```

The `Minimize` function works algebraically, which means it sometimes doesn't do quite what you'd like. It generally works well on polynomial problems, but if you give it a nasty enough trancendental problem, you're out of luck.

```
Minimize[Exp[-x^2] + Exp[(x - 2)^2], x]
```

```
Minimize[ℯ^(-2+x)^2 + ℯ^-x^2, x]
```

So, sometimes you get better results working just numerically. For that, try `NMinimize` .

```
NMinimize[ℯ^(-2+x)^2 + ℯ^-x^2, x]
```

```
{1.01712, {x → 2.03261}}
```

Here's our linear least squares fit again:

```
NMinimize[LeastSquaresError[data, lineFunction], {a, b}]
```

```
{15.3329, {a → 1.9935, b → 3.05974}}
```

Another numerical function is `FindMinimum`, which uses a different numerical algorithm. You have to specify a starting point for each unknown variable.

```
FindMinimum[LeastSquaresError[data, lineFunction], {{a, 1}, {b, 1}}]
```

```
{15.3329, {a → 1.9935, b → 3.05974}}
```

The `Fit` function does exactly this same minimization conceptually, but it only works if the fit function looks like

$$f[x] = a_1 f_1[x] + a_2 f_2[x] + \ldots + a_n f_n[x]$$

and the a_1, a_2, ... a_n are the only unknowns. This is the traditional method of curve fitting (predating modern computers that can do more powerful techniques almost as fast) because if f has this form, you can take a short cut from linear algebra and do the computation very quickly. Otherwise, the minimization can be computationally intensive and may get stuck at a local minimum instead of finding the global minimum.

Non−linear least squares

■ The linearization method

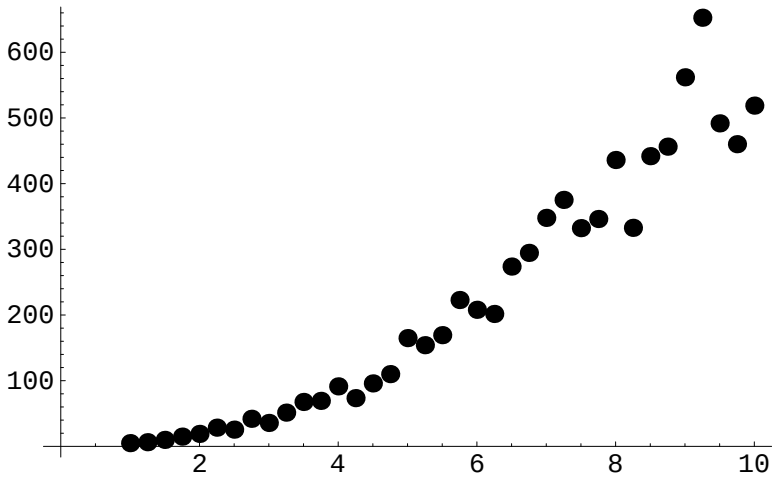
To do non−linear curve fitting with least squares, there are a couple of alternatives. One is to linearize the data first, then proceed using Fit.

As before, let's make up some noisy data to play with. It's basically $4\,x^{2.2}$ with noise added to the coefficient and the exponent.

```
nonlinearData = Table[{x,
   (4 + Random[Real, {-1, 1}]) x^(2.2 + Random[Real, {-0.05, 0.05}])},
  {x, 1, 10, 0.25}]
```

```
{{1, 4.92824}, {1.25, 6.35181}, {1.5, 9.8858},
 {1.75, 14.8502}, {2., 18.9945}, {2.25, 28.5018},
 {2.5, 25.4874}, {2.75, 41.9887}, {3., 35.6793},
 {3.25, 51.3869}, {3.5, 67.6365}, {3.75, 69.2924}, {4., 91.3692},
 {4.25, 73.4505}, {4.5, 95.8571}, {4.75, 110.006}, {5., 164.822},
 {5.25, 154.05}, {5.5, 169.333}, {5.75, 222.831}, {6., 207.939},
 {6.25, 201.619}, {6.5, 273.774}, {6.75, 294.59}, {7., 347.873},
 {7.25, 375.263}, {7.5, 332.254}, {7.75, 346.179}, {8., 436.014},
 {8.25, 332.709}, {8.5, 441.831}, {8.75, 456.313}, {9., 561.909},
 {9.25, 652.596}, {9.5, 491.78}, {9.75, 460.075}, {10., 518.81}}
```

```
nonlinearPointPlot =
 ListPlot[nonlinearData, PlotStyle → PointSize[0.025]]
```




```
- Graphics -
```

We'd like to fit this to a power function and find a and b.

```
powerFunction[x_] = a x^b
```

```
a x^b
```

But we can't use Fit, because the unknowns aren't in the right place. So, we linearize the data first. Assuming the power function is correct for our data, we can take the logarithm, and get something in the right form for Fit:

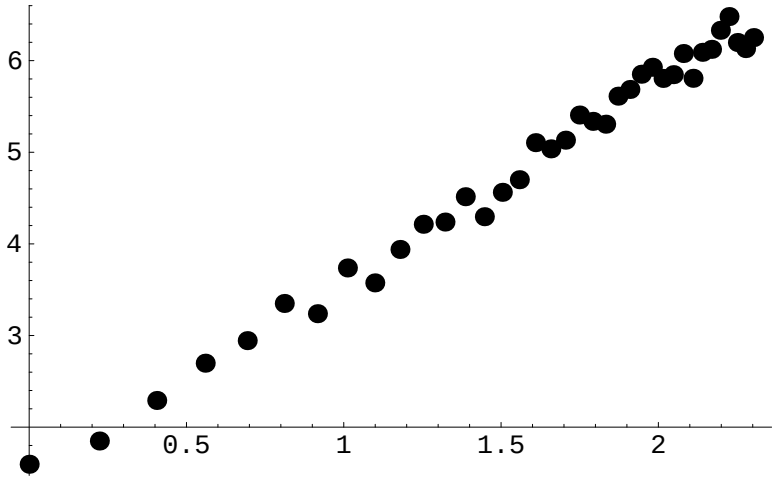
$$\text{Log}[y] = \text{Log}[a\, x^b] = \text{Log}[a] + \text{Log}[x^b] = \text{Log}[a] + b\, \text{Log}[x]$$

So even though our (x, y) data is not in the right form for Fit, it turns out that (Log[x], Log[y]) is in the right form. Here's an incantation to linearize the data. The trick is that `/.` can apply rules that involve patterns (see the *Mathematica* book 2.5), so this next command looks at nonlinearData and replaces anything that looks like $\{x, y\}$ with {Log[x], Log[y]}. As in function definitions, the _ on the x and y indicates that these are pattern variables. Without the _, *Mathematica* will think you mean to replace only stuff with the symbols x and y. And you don't use the _ on the right hand side of the rule or in a function definition, only on the left.

```
linearizedData = nonlinearData /. {x_, y_} → {Log[x], Log[y]}
```

```
{{0, 1.59498}, {0.223144, 1.84874},
 {0.405465, 2.2911}, {0.559616, 2.69801},
 {0.693147, 2.94415}, {0.81093, 3.34997}, {0.916291, 3.23818},
 {1.0116, 3.7374}, {1.09861, 3.57457}, {1.17865, 3.93938},
 {1.25276, 4.21415}, {1.32176, 4.23834}, {1.38629, 4.51491},
 {1.44692, 4.29661}, {1.50408, 4.56286}, {1.55814, 4.70053},
 {1.60944, 5.10487}, {1.65823, 5.03727}, {1.70475, 5.13187},
 {1.7492, 5.40641}, {1.79176, 5.33724}, {1.83258, 5.30638},
 {1.8718, 5.6123}, {1.90954, 5.68558}, {1.94591, 5.85184},
 {1.981, 5.92763}, {2.0149, 5.8059}, {2.04769, 5.84696},
 {2.07944, 6.07767}, {2.11021, 5.80727}, {2.14007, 6.09093},
 {2.16905, 6.12318}, {2.19722, 6.33134}, {2.22462, 6.48096},
 {2.25129, 6.19803}, {2.27727, 6.13139}, {2.30259, 6.25154}}
```

```
linearizedPointPlot =
 ListPlot[linearizedData, PlotStyle → PointSize[0.025]]
```



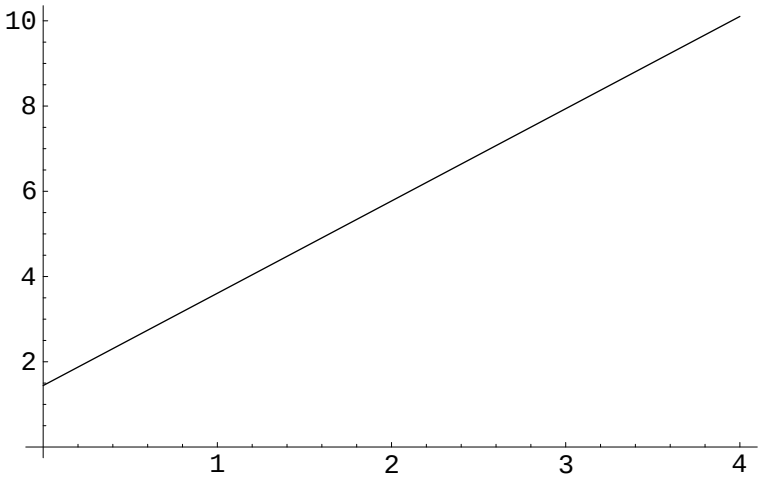

```
- Graphics -
```

Now we can use Fit:

```
logFit = Fit[linearizedData, {1, logx}, logx]
```

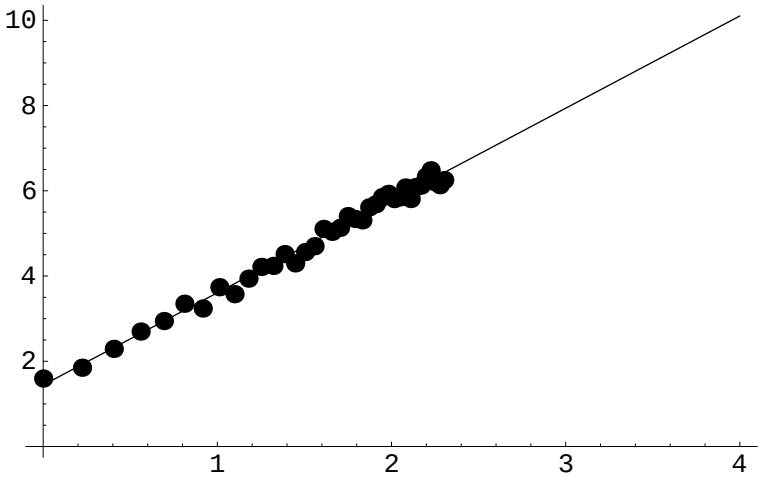
```
1.44227 + 2.16518 logx
```

```
logFitPlot = Plot[logFit, {logx, 0, 4}]
```




```
- Graphics -
```

```
Show[logFitPlot, linearizedPointPlot]
```



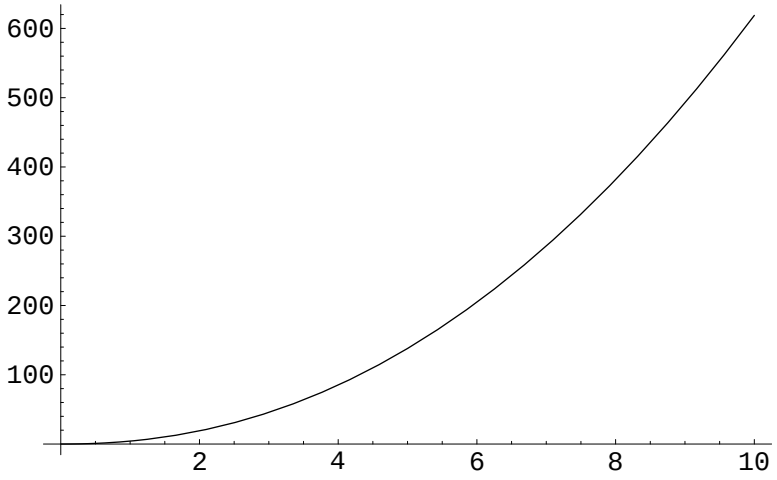

```
- Graphics -
```

And you can see that this is pretty close to $4\,x^{2.2}$:

```
nonlinearFit[x_] = Exp[logFit /. logx → Log[x]]
```

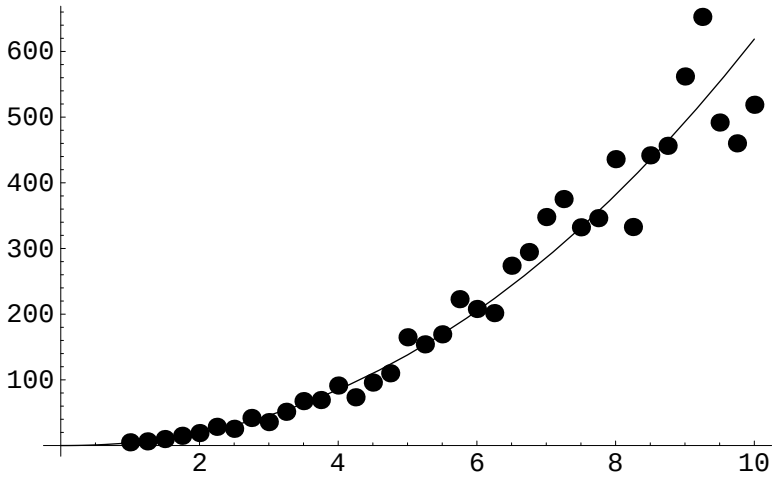
$$4.2303\,x^{2.16518}$$

```
nonlinearFitPlot = Plot[nonlinearFit[x], {x, 0, 10}]
```




```
- Graphics -
```

```
Show[nonlinearFitPlot, nonlinearPointPlot]
```




```
- Graphics -
```

■ The direct least squares method

Since *Mathematica* can directly minimize various expressions, we can also skip the linearization step and minimize the error directly:

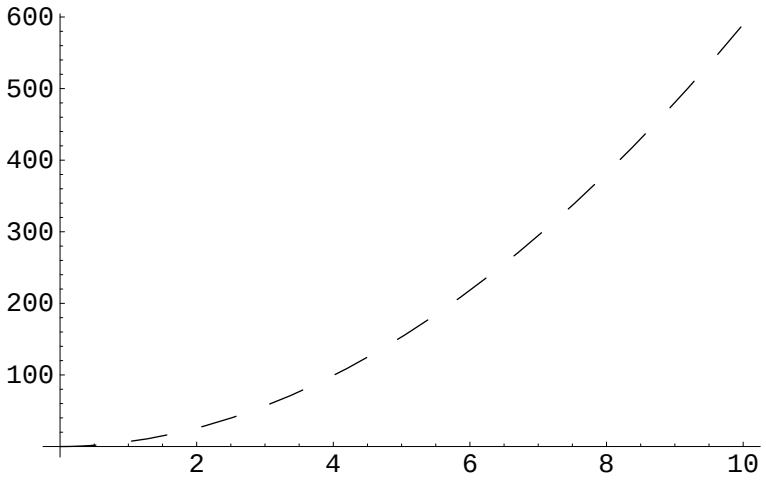
```
directFitSolution =
 Minimize[LeastSquaresError[nonlinearData, powerFunction], {a, b}]
```

```
{67653.3, {a → 6.67436, b → 1.94624}}
```

```
directFit[x_] = powerFunction[x] /. directFitSolution[[2]]
```

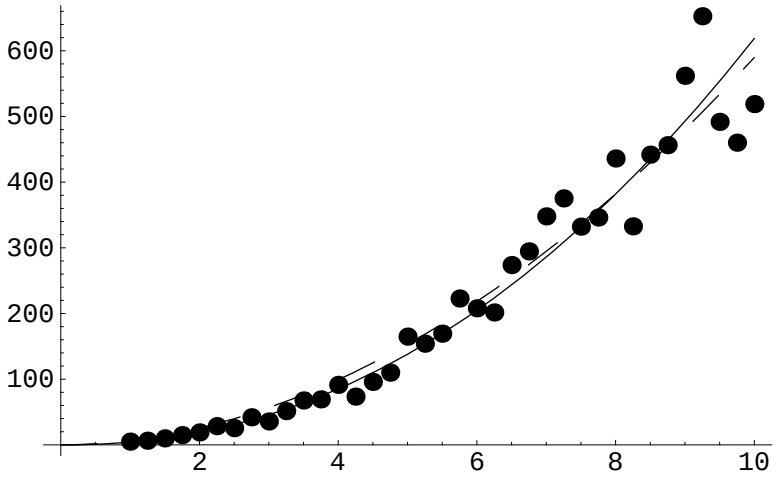
$$6.67436\, x^{1.94624}$$

```
directFitPlot = Plot[directFit[x], {x, 0, 10},
  PlotStyle → Dashing[{0.05, 0.05}]]
```



- Graphics -

```
Show[directFitPlot, nonlinearFitPlot, nonlinearPointPlot]
```



- Graphics -

And you should be able to see that the function found by the linearization method isn't quite the same as the one found by the direct method:

```
{nonlinearFit[x], directFit[x]}
```

$$\{4.2303\, x^{2.16518},\ 6.67436\, x^{1.94624}\}$$

Both functions are actually the optimum fit to the data, but under different notions of distance. And notice that neither one is exact compared to what we started with. For example, here's what I got on one run of this worksheet:

$\{3.5498\, x^{2.27148},\ 3.0792\, x^{2.34363}\}$

Since the data set is constructed with random noise, the results will be a little different each time you run it. But neither of these gives back $4\, x^{2.2}$ exactly. (Think about it: Should they?) And which curve is "better"?

This contention between the results illustrates the fundamental conceptual problems in curve fitting: (1) Guess the appropriate form of the function. (2) Determine an appropriate notion of "distance" between the curve and the data to minimize.

Maximum likelihood

An alternative notion of distance that is appropriate for modeling changing probabilities is likelihood. This notion assumes that the data is of the form $y_j = f(x_j, \omega_j; a, b, \ldots)$ where x is an independent variable, such as distance or time, ω_j are independent random numbers, and a, b, ... are the parameters that we want to find. Then, the likelihood of the data is the probability of getting exactly the y_j's. The curve fit procedure is to determine values of the parameters such that the likelihood of the data is maximal.

As an example, let's suppose we have a biological experiment that succeeds or fails, for example, getting bacteria to accept a fragment of DNA. Let's suppose that the probability of success depends on temperature x. Furthermore, let's suppose that we have some knowledge of the biochemistry involved that tells us that the probability of success is actually an exponential function: $p[x] = e^{-a(x-b)}$ where a and b are unknown. We are given results from experiments run at different temperatures, and they are simply given as success or failure, and our job is to find a and b.

First, let's invent some data, using $a = 0.025$ and $b = -5$.

```
pSuccess[x_, a_, b_] = Exp[-a (x - b)]
```

$e^{-a(-b+x)}$

`Random` returns a random number between 0 and 1, so the test **`Random[] < p`** returns True with probability p and False with probability $1 - p$, so we can use it to simulate our experiment. We'll also assume that this experiment is fairly expensive and time consuming, so we can't run zillions of experiments.

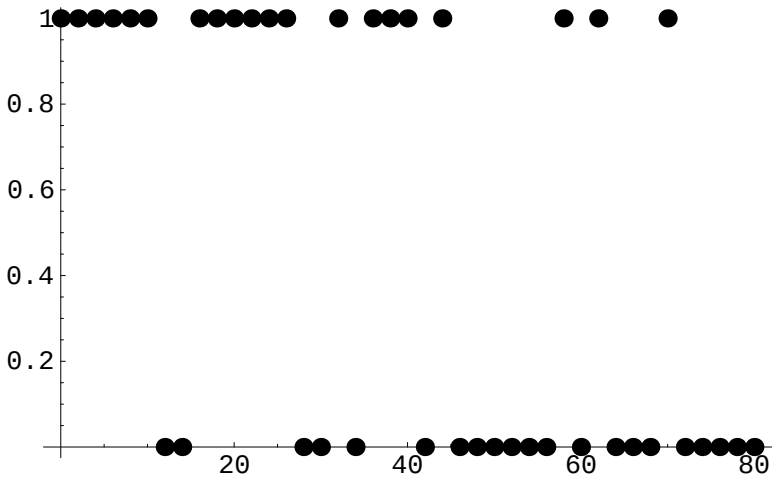
```
runExperiment[x_, a_, b_] := (Random[] < pSuccess[x, a, b])

bioExampleData = Table[{x, runExperiment[x, 0.025, -5]},
  {x, 0, 80, 2}]
```

```
{{0, True}, {2, True}, {4, True}, {6, True}, {8, True}, {10, True},
 {12, False}, {14, False}, {16, True}, {18, True}, {20, True},
 {22, True}, {24, True}, {26, True}, {28, False}, {30, False},
 {32, True}, {34, False}, {36, True}, {38, True}, {40, True},
 {42, False}, {44, True}, {46, False}, {48, False}, {50, False},
 {52, False}, {54, False}, {56, False}, {58, True}, {60, False},
 {62, True}, {64, False}, {66, False}, {68, False}, {70, True},
 {72, False}, {74, False}, {76, False}, {78, False}, {80, False}}
```

Here's a way to plot that data. Do you see how this command works?

```
ListPlot[bioExampleData /. {True → 1, False → 0},
 PlotStyle → PointSize[0.025]]
```




```
- Graphics -
```

The data we have isn't points (x, $p[x, a, b]$) which is what we'd need to do traditional curve fitting. In other words, we want to fit a curve but we don't have points from the curve plus noise like we did in earlier examples; we have something else entirely. Intuitively, it seems impossible for least squares to give anything useful for this type of data, so instead, we do maximum likelihood. The experiments are independent, so the probability of getting all this data is the product of the probabilities of getting each point. But the probability of getting each point depends on a and b.

```
likelihood[data_, a_, b_] := Product[
  If[data[[j, 2]],
   pSuccess[data[[j, 1]], a, b],
   1 - pSuccess[data[[j, 1]], a, b]],
  {j, 1, Length[data]}]
```

So for example:

```
likelihood[bioExampleData, 0.001, -4]
```

$$3.212 \times 10^{-28}$$

```
likelihood[bioExampleData, 0.02, 2]
```

$$3.57 \times 10^{-10}$$

These are tiny numbers, and that causes trouble with the numerical maximization process, so instead of maximizing the likelihood, we maximize the log likelihood (Think about it: Why can we do this?)

We could do this, but then *Mathematica* will compute that tiny likelihood, then take the log.

```
logLikelihood1[data_, a_, b_] := Log[likelihood[data, a, b]]
```

A better way to compute the same thing is to expand the log of the product into a sum of logs. I'll do this computation using different notation:

```
logLikelihood[data_, a_, b_] := Total[
  data /. {x_, success_} → If[success,
     Log[pSuccess[x, a, b]],
     Log[1 - pSuccess[x, a, b]]]]
```

Just to check that we did this right, these should be the same number:

```
Log[likelihood[bioExampleData, 0.001, -4]]
```

```
-63.3055
```

```
logLikelihood[bioExampleData, 0.001, -4]
```

```
-63.3055
```

Here's the first try at running the fit:

```
Maximize[{logLikelihood[bioExampleData, a, b], a > 0}, {a, b}]
```

```
Maximize[
 {Log[ℯ^-a (2-b) ] + Log[ℯ^-a (4-b) ] + Log[ℯ^-a (6-b) ] + Log[ℯ^-a (8-b) ] + Log[ℯ^-a (10-b) ] +
   Log[ℯ^-a (16-b) ] + Log[ℯ^-a (18-b) ] + Log[ℯ^-a (20-b) ] + Log[ℯ^-a (22-b) ] +
   Log[ℯ^-a (24-b) ] + Log[ℯ^-a (26-b) ] + Log[ℯ^-a (32-b) ] + Log[ℯ^-a (36-b) ] +
   Log[ℯ^-a (38-b) ] + Log[ℯ^-a (40-b) ] + Log[ℯ^-a (44-b) ] + Log[ℯ^-a (58-b) ] +
   Log[ℯ^-a (62-b) ] + Log[ℯ^-a (70-b) ] + Log[ℯ^a b ] + Log[1 - ℯ^-a (12-b) ] +
   Log[1 - ℯ^-a (14-b) ] + Log[1 - ℯ^-a (28-b) ] + Log[1 - ℯ^-a (30-b) ] +
   Log[1 - ℯ^-a (34-b) ] + Log[1 - ℯ^-a (42-b) ] + Log[1 - ℯ^-a (46-b) ] +
   Log[1 - ℯ^-a (48-b) ] + Log[1 - ℯ^-a (50-b) ] + Log[1 - ℯ^-a (52-b) ] +
   Log[1 - ℯ^-a (54-b) ] + Log[1 - ℯ^-a (56-b) ] + Log[1 - ℯ^-a (60-b) ] +
   Log[1 - ℯ^-a (64-b) ] + Log[1 - ℯ^-a (66-b) ] + Log[1 - ℯ^-a (68-b) ] +
   Log[1 - ℯ^-a (72-b) ] + Log[1 - ℯ^-a (74-b) ] + Log[1 - ℯ^-a (76-b) ] +
   Log[1 - ℯ^-a (78-b) ] + Log[1 - ℯ^-a (80-b) ], a > 0}, {a, b}]
```

Which means it couldn't solve the problem algebraically. So, let's try the numerical methods:

```
NMaximize[logLikelihood[bioExampleData, a, b], {a, b}]
```

```
NMaximize::nrnum : The function value -1524.76 - <<18>> i
   is not a real number at {a, b} = {-0.936293, <<20>>}. More...
```

```
NMaximize[Log[ℯ^-a (2-b) ] + Log[ℯ^-a (4-b) ] + Log[ℯ^-a (6-b) ] +
  Log[ℯ^-a (8-b) ] + Log[ℯ^-a (10-b) ] + Log[ℯ^-a (16-b) ] + Log[ℯ^-a (18-b) ] +
  Log[ℯ^-a (20-b) ] + Log[ℯ^-a (22-b) ] + Log[ℯ^-a (24-b) ] + Log[ℯ^-a (26-b) ] +
  Log[ℯ^-a (32-b) ] + Log[ℯ^-a (36-b) ] + Log[ℯ^-a (38-b) ] + Log[ℯ^-a (40-b) ] +
  Log[ℯ^-a (44-b) ] + Log[ℯ^-a (58-b) ] + Log[ℯ^-a (62-b) ] + Log[ℯ^-a (70-b) ] +
  Log[ℯ^a b ] + Log[1 - ℯ^-a (12-b) ] + Log[1 - ℯ^-a (14-b) ] + Log[1 - ℯ^-a (28-b) ] +
  Log[1 - ℯ^-a (30-b) ] + Log[1 - ℯ^-a (34-b) ] + Log[1 - ℯ^-a (42-b) ] + Log[1 - ℯ^-a (46-b) ] +
  Log[1 - ℯ^-a (48-b) ] + Log[1 - ℯ^-a (50-b) ] + Log[1 - ℯ^-a (52-b) ] + Log[1 - ℯ^-a (54-b) ] +
  Log[1 - ℯ^-a (56-b) ] + Log[1 - ℯ^-a (60-b) ] + Log[1 - ℯ^-a (64-b) ] + Log[1 - ℯ^-a (66-b) ] +
  Log[1 - ℯ^-a (68-b) ] + Log[1 - ℯ^-a (72-b) ] + Log[1 - ℯ^-a (74-b) ] +
  Log[1 - ℯ^-a (76-b) ] + Log[1 - ℯ^-a (78-b) ] + Log[1 - ℯ^-a (80-b) ], {a, b}]
```

You probably got an error message, either that an overflow occurred, or that it got a complex number somewhere along the way. That means that `NMaximize` is trying values of *a* and *b* that yield logs of negative numbers, or perhaps log of 0. Let's use the constraint feature of `NMaximize` to give it some additional hints: We ask it to

maximize the log likelihood, but subject to some reasonable constraints. We know the exponential should decrease as x increases, so we add in $a > 0$.

```
NMaximize[{logLikelihood[bioExampleData, a, b], a > 0}, {a, b}]
```

```
{-21.2748, {a → 0.0277435, b → 7.11101}}
```

Sometimes that works, sometimes it doesn't, depending on what random numbers appeared in our simulated experiment. Let's try `FindMaximum` since it takes a starting point.

```
FindMaximum[logLikelihood[bioExampleData, a, b], {{a, 0}, {b, -1}}]
```

```
FindMaximum::nnum :
 The function value Indeterminate is not a number at {a, b} = {0., -1.}. More...
```

```
FindMaximum[logLikelihood[bioExampleData, a, b], {{a, 0}, {b, -1}}]
```

That probably didn't work because we get something like Log[0] somewhere if we start at $a = 0$.

```
FindMaximum[logLikelihood[bioExampleData, a, b], {{a, 0.01}, {b, -1}}]
```

```
FindMaximum::lstol :
 The line search decreased the step size to within tolerance specified by
   AccuracyGoal and PrecisionGoal but was unable to find a sufficient
   increase in the function.  You may need more than MachinePrecision
   digits of working precision to meet these tolerances. More...
```

```
{-21.2748, {a → 0.0277435, b → 7.11101}}
```

This is pretty good, at least the time I ran it. Sometimes you have to poke around with different starting points to get reasonable results. There are also zillions of options, and you can spend lots of time playing with them, tweaking the numerical method, but unless you're desperate or know what all the tweaks mean, doing that is often a waste of time.

The result isn't perfect, but it has a definite interpretation: These values for a and b are the ones such that the probability of getting our observations is maximal. And they're reasonably close to the exact answer, which is great given how little information our data acutally contains.