Analyzing functions: lab 3

©2005 Ben Bolker, modified at some places by Alejandro Morales & Ioannis Baltzakis 2017

November 3, 2017

This lab will be somewhat shorter in terms of "`R` stuff" than the previous labs, because more of the new material is algebra and calculus than `R` commands. Try to do a reasonable amount of the work with paper and pencil before resorting to messing around in `R`.

derivatives ifelse for thresholds

1 For loops

When programming your data analysis, you often need to iterate over multiple elements of a collection. These elements could be rows in a `data.frame`, datasets inside a `list`, numbers inside a sequence, etc. The iteration usually means that you apply the same code over each element of the collection and you don't want to "copy-paste" the code for each element. In programming, an iteration over a collection is called a "for loop". A for loop consists of three components:

1. A collection over which you want to iterate.
2. A variable that keeps track of where you are in the collection in each iteration
3. The body of the loop where you apply some code.

Imagine that you want to calculate the factorial of 10. The factorial of a number is simply the product of all positive numbers smaller or equal than the number you specify (and it is denote with a "!" after the number). For example, the factorial of 3 is $3! = 3 \times 2 \times 1$. A simple way of calculating the factorial of 10 is

```
result = 1
for(i in 1:10) {
  result = result*i
}
```

In this for loop, the collection is `1:10`, the variable to keep track of the number is `i` and the body of the loop is simply `result = result*i`. This for loop shows a very typical pattern: we want to summarise some collection of numbers into a single number, in this case, `result`. Another typical pattern is when we want to update the elements of a collection. In this case, it is a good practice to "pre-allocate" your output collection before looping. For example,

```r
x = 1:10
y = numeric(10)
for(i in 1:10) {
  y[i] = exp(x[i])
}
```

In thise case, the result of the for loop (`y`) is a collection of 10 elements where each element is the exponential transformation of the element in `x` with the same position. Note that we specify before the loop that `y` will have 10 elements. Although this is not strictly required in this case, it is a good practice both to avoid errors and to make your code run faster.

In practice, for loops are not that common in `R` as in other languages The reason is that many mathematical and statistical functions are already, implicitly, looping over your collections. For examples, when you take the exponential (`exp()`) of a collection of numbers, it will produce a new collection which is the result of looping over the original collection. That is:

```r
x = 1:10
y = exp(x)
```

is equivalent to the previous loop described before. As you can see, this second option requires less code and it is easier to read, which is one of the reasons why `R` is such a greate language for working with data. In addition, if you rely on this implicit looping your code will run much faster.

However, there may be situations where you really cannot avoid a for loop. For example, if you have collected multiple datasets and need to perform the same analysis on each dataset, you could store your datasets in a list and use a for loop to iterate over the different datasets.

Exercise 1.1: Calculate the logarithm of the sequence 1:10, using first a for loop and then without a for loop.

2 Missing values (NA)

The `R` languages has a special way of representing missing values in a dataset. A missing value is denoted with the symbol `NA` which stands for "**N**ot **A**vailable". By default, missing values will "propagate" throughout the calculations. For example, given two vectors of data:

```r
x = c(1,2,3)
y = c(2,4,NA)
```

When you combine these vectors (e.g. add them or multiply them) you will see that the third component is always `NA`

```r
x + y
```

```
## [1]  3  6 NA
```

```r
x*y
```

```
## [1]  2  8 NA
```

When you calculate some statistical property of your data (e.g. mean, standard deviation) it will, by default, report `NA` if there is at least one missing value in your data

```r
mean(x)
```

```
## [1] 2
```

```r
mean(y)
```

```
## [1] NA
```

Most statistical functions in `R` allow you to specify how to deal with missing values. Most often, you are given the option to ignore any missing values from the data when calculating an statistical property through an argument often called `na.rm`. For example, in order to get the mean of the non-missing values of `y` we need:

```r
mean(y, na.rm = TRUE)
```

```
## [1] 3
```

which, of course, is the mean of 2 and 4. However, other functions will not have an option to handle `NA` even though you still need to make a decision on how to deal with them. For example, when you calculate the length of a dataset (`length()`) do you want to consider the whole data or only the non-missing values? This is not a trivial question and the answer on the context where you will use the result. In any case, if you want to remove the `NA` when calculating the length you need to be more creative. Fortunately, `R` offers the function `is.na` which tells which data points are missing in a dataset, so you could this:

```r
length(y[!is.na(y)])
```

```
## [1] 2
```

Which only gives 2 as the third element is missing. Remember that `!` is a negation operator, so `!is.na` actually means “is not NA”.

By the way, you should not confuse `NA` with `NaN` which stands for “**N**ot **A** **N**umber”. An `NaN` is the result of either an expression with indeterminate form (e.g. `0/0` or `Inf/Inf`) or when a function is evaluated outside of its valid domain (e.g. `sqrt(-1)` or `log(-1)`).

Exercise 2.1: Given some data created from the following code `c(25,1,10,89, NA, NA)`, calculate the mean value and the standard error of this mean ($s.e.m. = \sigma/\sqrt{n}$, where σ is the standard deviation and n is the amount of data) by ignoring missing values.

3 How to make a function

When you want to repeated some calculations for different data, it is best to put the data inside a function. A function is defined by 4 elements

1. The name of the function. For example, in `R` there is a function that calculates the arithmetic mean of a vector of data and its name is `mean`. You should make sure that the name of your function does not coincide with existing functions, that it is not too long and that it conveys its meaning.
2. The arguments of the function. These are the variables that you need to pass to the function (i.e., inputs). The arguments are defined by a position and a name. Also, some arguments may have default values which means that you do not need to specify them every time you call the function. FOr example, the function `mean`, contains three arguments (`x`, `trim` and `na.rm`) but the last two have default values.
3. The body of the function. This is the actual code that the function will execute. The real `mean` function in R has some crytpic body that requires advanced knowledge of the language to understand. However, a more “naive” implementation of `mean` could be `sum(x)/length(x)`. Note that the body of a function can consiste of multiple lines.
4. The return value of the function. This is the result of applying the function on the arguments. By default, the result of the last line code in the body of the function is the return value of the function. You can also return from the at any point in the body with the function `return()` with the variable you want to return inside.

The `R` language specifies a particular syntax on how to build a function. For example, a `naive_mean` could be defined as:

```r
naive_mean = function(x, na.rm = TRUE) {
  total = sum(x, na.rm = na.rm)
  n = length(!is.na(x))
  result = total/n
  return(result)
}
```

In this case, there is are two arguments (`x` and `na.rm`) where the second argument has a default value and the body consiste of several lines of code, with the last one returning the result. Notice that arguments are separated by commas and the body of the function is enclosed in curly braces `{}`. The name of the function is simply the name of the variable to which you assigned the function (i.e., `naive_mean`). You can see below that you can use this function in a similar manner to the built-in `mean`

```r
x = 1:10
naive_mean(x)
```

```
## [1] 5.5
```

Notice that we did not specify the value of `na.rm` as the default is ok in this case. However, if we had missing values, the NA would propagate to the output:

```r
x = c(1,2,NA,4)
naive_mean(x)
```

```
## [1] 1.75
```

Which forces us to make a decision. Let's say that, for the moment, we want to just remove the values that are NA from the calculation. In this case, we just change the value of the default parameter.

```r
naive_mean(x, na.rm = TRUE)
```

```
## [1] 1.75
```

For convenience, default parameters are specified by name rather than position. However we could have also said `naive_mean(x,TRUE)` or even `naive_mean(x = x, na.rm = TRUE)`. All these forms of calling functions are OK, whether you choose one style or another is a matter of taste.

Exercise 3.1: Build a function to calculate the standard deviation ($\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$). Test your function with some data that includes missing values.

4. Numerical experimentation: plotting curves

Here are the R commands used to generate Figure 1. They just use curve(), with add=FALSE (the default, which draws a new plot) and add=TRUE (adds the curve to an existing plot), particular values of from and to, and various graphical parameters (ylim, ylab, lty).

```
curve(2*exp(-x/2),from=0,to=7,ylim=c(0,2),ylab="")
curve(2*exp(-x),add=TRUE,lty=4)
curve(x*exp(-x/2),add=TRUE,lty=2)
curve(2*x*exp(-x/2),add=TRUE,lty=3)
text(0.4,1.9,expression(paste("exponential: ",2*e^(-x/2))),adj=0)
text(4,.5,expression(paste("Ricker: ",x*e^(-x/2))))
text(4,1,expression(paste("Ricker: ",2*x*e^(-x/2))),adj=0)
text(2.8,0,expression(paste("exponential: ",2*e^(-x))))
```

The only new thing in this figure is the use of expression() to add a mathematical formula to an R graphic. text(x,y,"x^2") puts x^2 on the graph at position (x, y); text(x,y,expression(x^2)) (no quotation marks) puts x^2 on the graph. See ?plotmath or ?demo(plotmath) for (much) more information.

An alternate way of plotting the exponential parts of this curve:

```
xvec = seq(0,7,length=100)
exp1_vec = 2*exp(-xvec/2)
exp2_vec = 2*exp(-xvec)
```

```
plot(xvec,exp1_vec,type="l",ylim=c(0,2),ylab="")
lines(xvec,exp2_vec,lty=4)
```

or, since both exponential vectors are the same length, we could cbind() them together and use matplot():

```
matplot(xvec,cbind(exp1_vec,exp2_vec),type="l",
        ylab="")
```

Finally, if you needed to use sapply() you could say:

```
expfun = function(x,a=1,b=1) {
   a*exp(-b*x)
 }
exp1_vec = sapply(xvec,expfun,a=2,b=1/2)
exp2_vec = sapply(xvec,expfun,a=2,b=1)
```

The advantage of curve() is that you don't have to define any vectors: the advantage of doing things the other way arises when you want to keep the vectors around to do other calculations with them.

Exercise 4.1 *: Construct a curve that has a maximum at ($x = 5$, $y = 1$). Write the equation, draw the curve in R, and explain how you got there.

4.1 A quick digression: `ifelse()` for piecewise functions

The `ifelse()` command in R is useful for constructing piecewise functions. Its basic syntax is `ifelse(condition,value_if_true,value_if_false)`, where `condition` is a logical vector (e.g. `x>0`), `value_if_true` is a vector of alternatives to use if `condition` is `TRUE`, and `value_if_false` is a vector of alternatives to use if `condition` is `FALSE`. If you specify just one value, it will be expanded (*recycled* in R jargon) to be the right length. A simple example:

```
x=c(-25,-16,-9,-4,-1,0,1,4,9,16,25)
ifelse(x<0,0,sqrt(x))
```

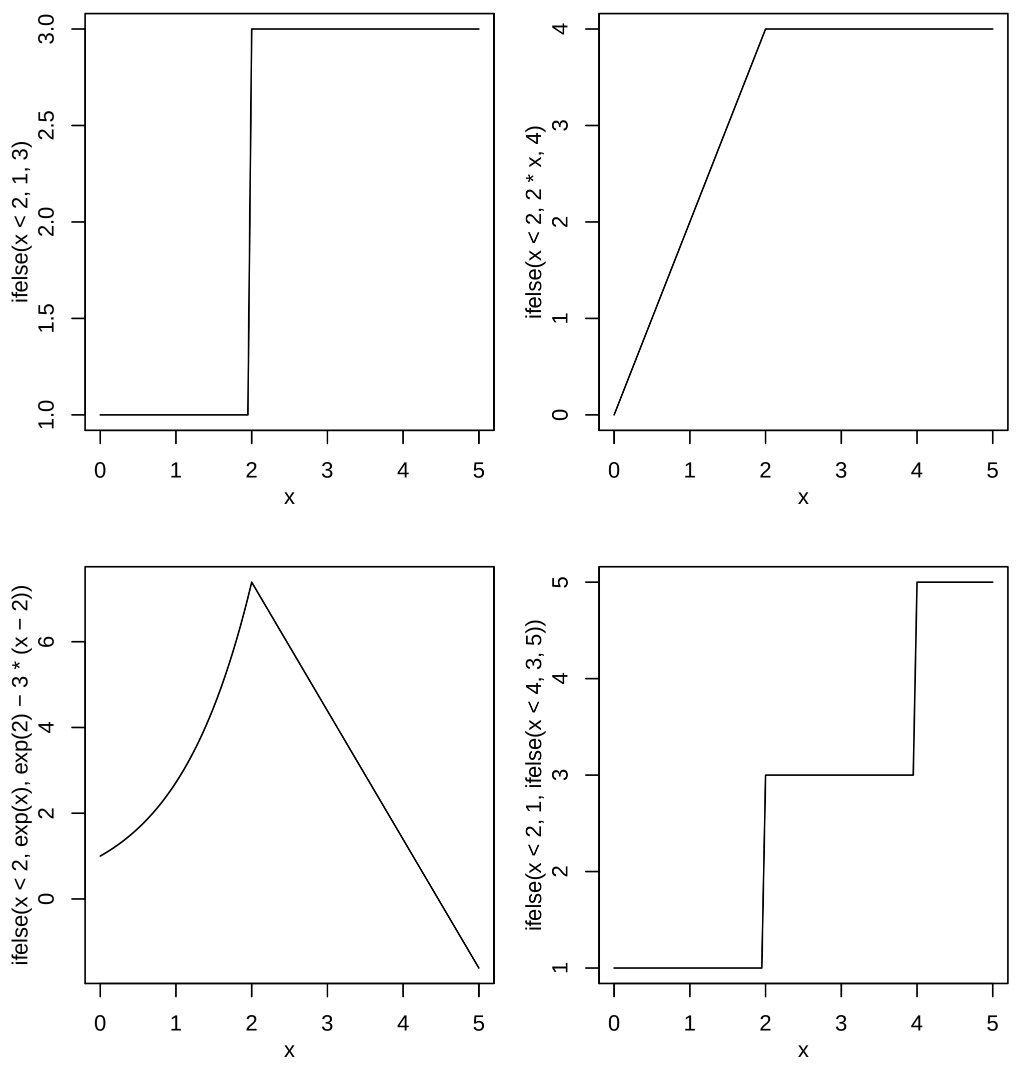
```
## Warning in sqrt(x): NaNs produced
```

```
##  [1] 0 0 0 0 0 0 1 2 3 4 5
```

These commands produce a warning message, but it's OK to ignore it since you know you've taken care of the problem (if you said `sqrt(ifelse(x<0,0,x))` instead you wouldn't get a warning: why not?)

Here are some examples of using `ifelse()` to generate (1) a simple threshold; (2) a Holling type I or "hockey stick"; (3) a more complicated piecewise model that grows exponentially and then decreases linearly; (4) a double-threshold model.

```
op=par(mfrow=c(2,2),mgp=c(2,1,0),mar=c(4.2,3,1,1))
curve(ifelse(x<2,1,3),from=0,to=5)
curve(ifelse(x<2,2*x,4),from=0,to=5)
curve(ifelse(x<2,exp(x),exp(2)-3*(x-2)),from=0,to=5)
curve(ifelse(x<2,1,ifelse(x<4,3,5)),from=0,to=5)
```



The double-threshold example (nested `ifelse()` commands) probably needs more explanation. In words, this command would go "if x is less than 2, set y to 1; otherwise ($x \geq 2$), if x is less than 4 (i.e. $2 \leq x < 4$), set y to 3; otherwise ($x \geq 4$), set y to 5".

5 Evaluating derivatives in `R`

`R` can evaluate derivatives, but it is not very good at simplifying them. In order for `R` to know that you really mean (e.g) `x^2` to be a mathematical expression and not a calculation for R to try to do (and either fill in the current value of x or give an error that x is undefined), you have to specify it as expression(x^2); you also have to tell R (in quotation marks) what variable you want to differentiate with respect to:

```
d1 = D(expression(x^2),"x"); d1
```

```
## 2 * x
```

Use eval() to fill in a list of particular values for which you want a numeric answer:

```
eval(d1,list(x=2))
```

```
## [1] 4
```

Taking the second derivative:

```
D(d1,"x")
```

```
## [1] 2
```

(As of version 2.0.1,) R knows how to take the derivatives of expressions including all the basic arithmetic operators; exponentials and logarithms; trigonometric inverse trig, and hyperbolic trig functions; square roots; and normal (Gaussian) density and cumulative density functions; and gamma and log-gamma functions. You're on your own for anything else (consider using a symbolic algebra package like Mathematica or Maple, at least to check your answers, if your problem is very complicated). deriv() is a slightly more complicated version of D() that is useful for incorporating the results of differentiation into functions: see the help page.

6 Figuring out the logistic curve

The last part of this exercise is an example of figuring out a function — Chapter 3 did this for the exponential, Ricker, and Michaelis-Menten functions The population-dynamics form of the logistic equation is

$$n(t) = \frac{K}{1 + \left(\frac{K}{n(0)} - 1\right) \exp(-rt)} \tag{1}$$

where K is carrying capacity, r is intrinsic population growth rate, and $n(0)$ is initial density.

At $t = 0$, $e^{-rt} = 1$ and this reduces to $n(0)$ (as it had better!)

Finding the derivative with respect to time is pretty ugly, but it will to reduce to something you may already know. Writing the equation as $n(t) = K \cdot (\text{stuff})^{-1}$ and using the chain rule we get $n'(t) = K \cdot (\text{stuff})^{-2} \cdot d(\text{stuff})/dt$ ($\text{stuff} = 1 + (K/n(0) - 1)\exp(-rt)$). The derivative $d(\text{stuff})/dt$ is $(K/n(0) - 1) \cdot -r\exp(-rt)$. At $t = 0$, $\text{stuff} = K/n(0)$, and $d(\text{stuff})/dt = -r(K/n(0) - 1)$. So this all comes out to

$$K \cdot (K/n(0))^{-2} \cdot -r(K/(n0) - 1) = -rn(0)^2/K \cdot (K/(n0) - 1) = rn(0)(1 - n(0)/K)$$

which should be reminiscent of intro. ecology: we have rediscovered, by working backwards from the time-dependent solution, that the logistic equation arises from a linearly decreasing *per capita* growth rate.

If $n(0)$ is small we can do better than just getting the intercept and slope.

Exercise 6.1* : show that if $n(0)$ is very small (and t is not too big), $n(t) \approx n(0)\exp(rt)$. (Start by showing that $K/n(0)e^{-rt}$ dominates all the other terms in the denominator.)

If t is small, this reduces (because $e^{rt} \approx 1 + rt$) to $n(t) \approx n(0) + rn(0)t$, a linear increase with slope $rn(0)$. Convince yourself that this matches the expression we got for the derivative when $n(0)$ is small.

For large t, convince yourself tha the value of the function approaches K and (by revisiting the expressions for the derivative above) that the slope approaches zero.

The half-maximum of the logistic curve occurs when the denominator (which I was calling "stuff" on the previous page, after eq. 1) is 2; we can solve $\text{stuff} = 2$ for t (getting to $(K/n(0) - 1)\exp(-rt) = 1$ and taking logarithms on both sides) to get $t = \log(K/n(0) - 1)/r$.

We have (roughly) three options:

1. Use `curve()`:

```r
r = 1
K = 1
n0 = 0.1  # what happens if you set n0 to 0???
curve(K/(1+(K/n0-1)*exp(-r*x)),from=0,to=10)
```

(note that we have to use `x` and not `t` in the expression for the logistic).

2. Construct the time vector by hand and compute a vector of population values using vectorized operations:

```r
t_vec = seq(0,10,length=100)
logist_vec = K/(1+(K/n0-1)*exp(-r*t_vec))
plot(t_vec,logist_vec,type="l")
```

3. write our own function for the logistic and use `sapply()`:

```
logistfun = function(t,r=1,n0=0.1,K=1) {
  K/(1+(K/n0-1)*exp(-r*t))
}
logist_vec = sapply(t_vec,logistfun)
```

(Setting e.g. `r=1` sets 1 as the default value for the parameter; if I omit some of the parameters when I call this function, `R` will fill in the default values.)

When we use this function, it will no longer matter how `r`, `n0` and `K` are defined in the workspace: the values that `R` uses in `logistfun()` are those that we define in the call to the function.

```
r=17
logistfun(1,r=2)
```

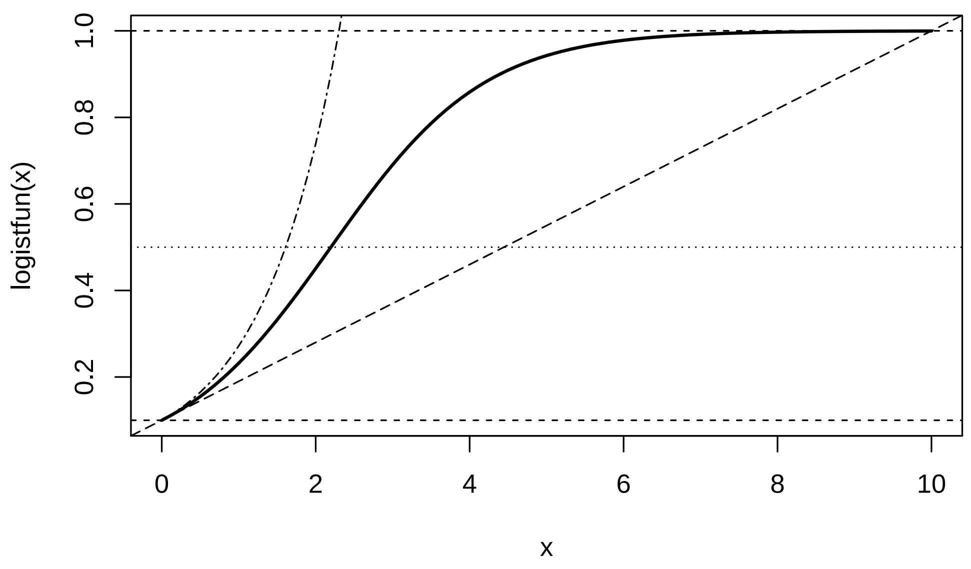
```
## [1] 0.4508531
```

```
r=0
logistfun(1,r=2)
```

```
## [1] 0.4508531
```

We can do more with this plot: let's see if our conjectures are right.
Using `abline()` and `curve()` to add horizontal lines to a plot to test our estimates of starting value and ending value, vertical and horizontal lines that intersect at the half-maximum, a line with the intercept and initial linear slope, and a curve corresponding to the initial exponential increase:

```
curve(logistfun(x),from=0,to=10,lwd=2)
abline(h=n0,lty=2)
abline(h=K,lty=2)
abline(h=K/2,lty=3)
abline(v=-log(n0/(K-n0))/r,lty=4)
r=1
abline(a=n0,b=r*n0*(1-n0/K),lty=5)
curve(n0*exp(r*x),from=0,lty=6,add=TRUE)
```



Exercise 6.2* : Plot and analyze the Shepherd function $G(N) = \frac{RN}{(1+aN)^b}$, which is a generalization of the Michaelis-Menten function. What are the effects of the R and a parameters on the curve? For what parameter values does this function become equivalent to the Michaelis-Menten function? What is the behavior (value, initial slope) at $N = 0$? What is the behavior (asymptote [if any], slope) for large N, for $b = 0$, $0 < b < 1$, $b = 1$, $b > 1$? Define an `R` function for the Shepherd function (call it `shep`). Draw a plot or plots showing the behavior for the ranges above, including lines that show the initial slope. Extra credit: when does the function have a maximum between 0 and ∞? What is the height of the maximum when it occurs? (Hint: when you're figuring out whether a fraction is zero or not, you don't have to think about the denominator at all.) The calculus isn't that hard, but you may also use the `D()` function in `R`. Draw horizontal and vertical lines onto the graph to test your answer.

Exercise 6.3 * : The Holling type~III functional response ($f(x) = ax^2/(1+bx^2)$) is useful when (e.g.) the predation rate initially accelerates with prey density, but then saturates. However, the parameters a (curvature at low prey density) and b (the reciprocal of the half-maximum squared) aren't easy to read off a graph. Reparameterize the Holling type~III function in terms of its asymptote and half-maximum.

Exercise 6.4* : Figure out the correspondence between the population-dynamic parameterization of the logistic function (eq. 1: parameters r, $n(0)$, K) and the statistical parameterization ($f(x) = \exp(a+bx)/(1+\exp(a+bx))$: parameters a, b). Convince yourself you got the right answer by plotting the logistic with $a = -5$, $b = 2$ (with lines), figuring out the equivalent values of K, r, and $n(0)$, and then plotting the curve with both equations to make sure it overlaps. Plot the statistical version with lines (`plot(...,type="l")} or`curve(...)} and then add the population-dynamic version with points (`points()} or`curve(...,type="p",add=TRUE)}).

Small hint: the population-dynamic version has an extra parameter, so one of r, $n(0)$, and K will be set to a constant when you translate to the statistical version.

Big hint: Multiply the numerator and denominator of the statistical form by $\exp(-a)$ and the numerator and denominator of the population-dynamic form by $\exp(rt)$, then compare the forms of the equations.