

May 18, 2020 – Hands-On Intro to the Basics of Scientific Programming Prof. Joshua Weitz and QBioS Cohort

Would you like to be able to build a computational model of a living system but don't know how? Then you are in the right spot.

This tutorial is meant to be interactive... that is *you* should be reading, typing, thinking, and asking questions. There are no source files to share, the point is for you to think about and enter the code and see the result. And then, to vary the code and see what changes. This is a more effective strategy than just *copying* code others have written and running it.

The materials are adapted from a semester long course entitled “Foundations of Quantitative Biosciences” developed by Prof. Joshua Weitz in Fall 2016/2017/2018/2019 as the cornerstone class for the QBioS Ph.D. at Georgia Tech. The materials have been adapted to focus on modelling epidemics and to account for the greater variety of backgrounds of students in this week's workshop.

Today we will focus on the basics of coding that can help you build models... whether of gene expression, cellular dynamics, game theory, or some other problem linked to dynamics of living systems. A few notes before we get started:

- If you are experienced in coding, then please work on challenge problems identified in this tutorial.
- Please choose one language - MATLAB, Python or R - and stick with it for the workshop.
- Please work on your own computer, save files for future reference, but also - work with a partner so you can discuss and help each other as you go.

Let's get started!

1 Getting Started

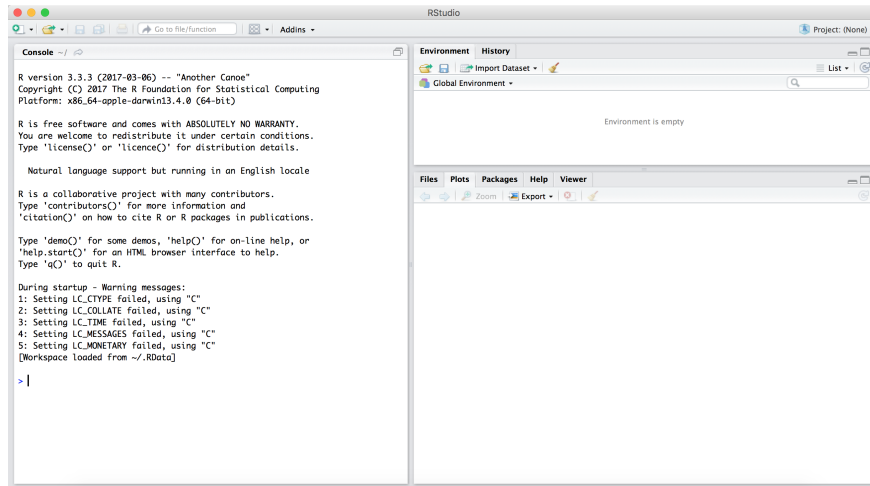
R is an open-source programming language for statistical computing and graphics. R is well known for its rich package ecosystem, as well as the ease with which one can produce good-quality plots and graphics. We will be using it in this class to simulate and model the dynamics of living systems from scales of molecules to ecosystems. R is particularly good at:

- Data manipulation
- Statistics
- Visualization
- Interfacing with other languages

and is okay at:

- Data & memory handling
- Speed (as compared to Python and Matlab)

There are a variety of Integrated Development Environments (IDEs) for R. For this course we will use RStudio. So, let's get started! In today's class, you will gain practical experience using R. **If you are uncertain how to launch R, try searching for "RStudio" using the search bar.** Then you should see a window that looks like this:



For an introduction to RStudio we recommend checking out this website: <https://datascienceplus.com/introduction-to-rstudio/>

We will be doing most of our work in the Console, the region to the bottom left. The Console is where you enter commands, and you should see a prompt that looks like this:

```
>
```

You can do basic math at this prompt (in order to run the cell press Enter), for example:

```
> 3+4
[1] 7
```

and

```
> exp(1)
[1] 2.718282
```

Alternatively, you can write your commands in the script file (top left) and press `ctrl + Enter` to run it in the console.

One of R's greatest strength is the diverse library of packages available to it. There are an incredible amount of packages that are useful for everything from 3D design to Genomics. Making use of this diverse set of packages requires that they be loaded into the current workspace. Installing and importing them is simple. Remember, that in every new session you must load the packages you wish to use. For example, to install the "fitdistrplus" package you can type in the console:

```
> install.packages("fitdistrplus")
```

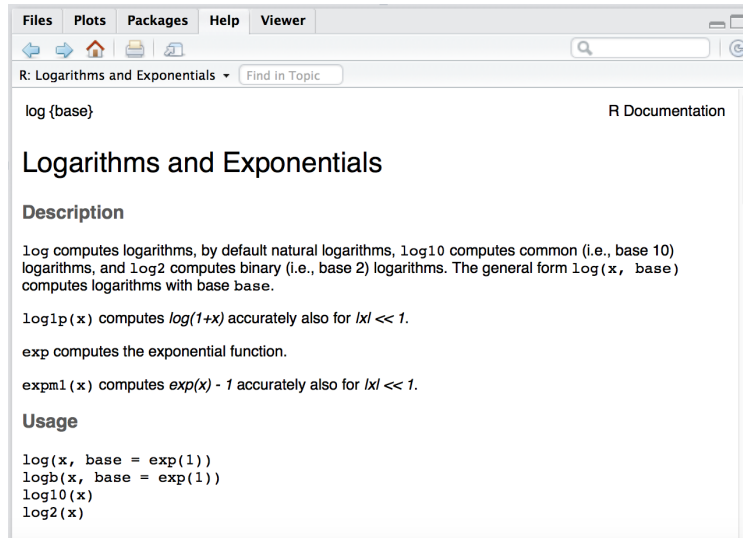
And then load it using:

```
> library("fitdistrplus")
```

Once a package is loaded, it is made available for every file opened in RStudio, so be careful when jumping between files. It is good practice to have import statements in the first few lines of every file. R has several functions that are built in (and you can use it like a calculator). For example, to learn more about the exponential function:

```
>?exp
```

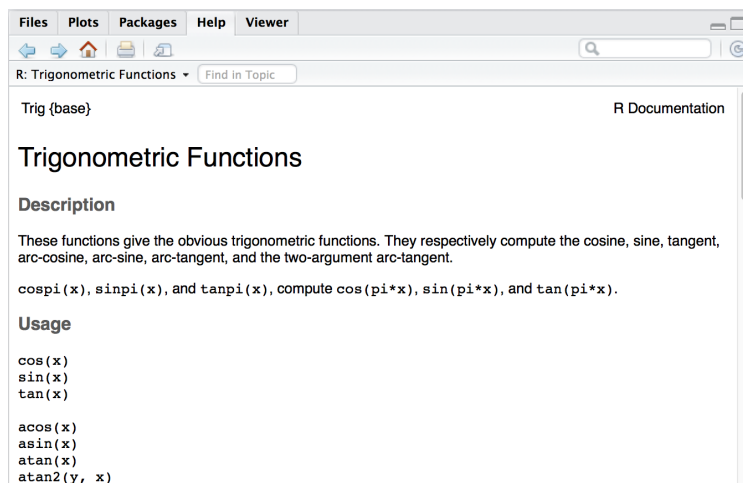
A help page should pop to your right. It looks like this:



For any function you can use `?function` or `help("function")`.

Many functions have names that you expect (how do you think you should calculate cosine or sine of a value, for example)? Try it out! If you don't know the name of the function you can use `??` followed by a key word.

```
> ??trigonometry
```



R is not just a calculator. It is also a programming language that can store values in memory. For example, the command:

```
> x <- 3
> x
[1] 3
```

tells R that the variable `x` has the value 3 and now every time you use “`x`”, R will substitute the value 3, for example:

```
> y = x + 1
> print(y)
[1] 4
```

Note that you can use either `=` or `<-` to assign values. If you don't want R to report back the answer/value you can either call the variable or use a `print()` function. It is very important to realize the `"="` sign does not mean that Python checks to see if the two sides are equal to each other, but rather states that whatever is on the left should be assigned the value of that on the right. Continuing our example from above, if you want to check the truth of a particular statement such as is `x` equal to 3, or alternatively, is `y` equal to 3 then you would type:

```
> x == 3
[1] TRUE

> y == 3
[1] FALSE
```

The double `"=="` sign tells R to logically compare what is on the left with that on the right and return 1 if true and 0 if false.

R can also handle arrays of values (such as vectors or matrices). The simplest way is to use the colon, which defines a sequential vector, for example:

```
> v <- 1:5
> print(v)
[1] 1 2 3 4 5
```

or you can use the `seq` function.

```
> z <- seq(1,5)
> print(z)
[1] 1 2 3 4 5
```

you can also modify the increments by adding a step parameter at the end

```
> w <- seq(1,9,2)
> print(w)
[1] 1 3 5 7 9
```

Any entry can be examined using the brackets

```
> w[3]
[1] 5
```

and basic math can be performed automatically on vectors (and matrices), for example

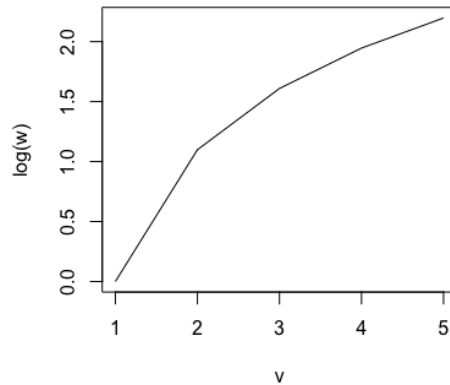
```
> log(w)
[1] 0.000000 1.098612 1.609438 1.945910 2.197225
```

R vectors/arrays are 1 indexed, meaning the first element's index is 1. Other languages like Python are 0 indexed, meaning the first element's index is 0 not 1.

R can also plot graphs and surfaces and all sorts of things. To create a simple plot, use the `plot` command.

```
> plot(v,log(w),type="l")
```

which leads to:



2 Building “Programs” from “Scripts” and “Functions”

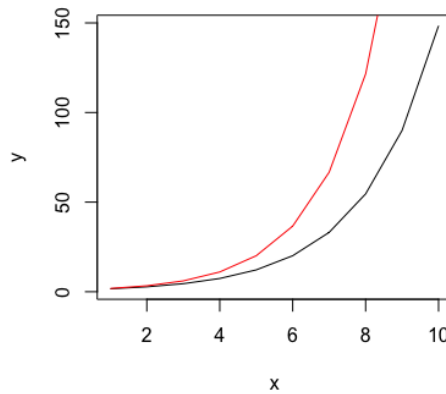
However, once you have a lot of commands, it will get exhausting typing them (especially when you make mistakes). So, R has functions, which is a list of commands, that execute in order. To create a function type in the panel on the top left of your screen and define a function using the command function.

```
# My first function
func <- function(){
  x <- 1:10
  y1 <- exp(x*.5)
  y2 <- exp(x*.6)
  plot(x,y1,type="l", xlab = "x", ylab= "y")
  lines(x,y2,col='red')
}
```

Run it, and then in the console type.

```
> func()
```

and it should do all the commands in your script and give you your figure.



The problem with this function is that if you wanted to change the exponents in these files, you would have to edit the function and then re-run it, instead of designating the change in the current line. Moreover, this function does not return a variable or accept a variable as input.

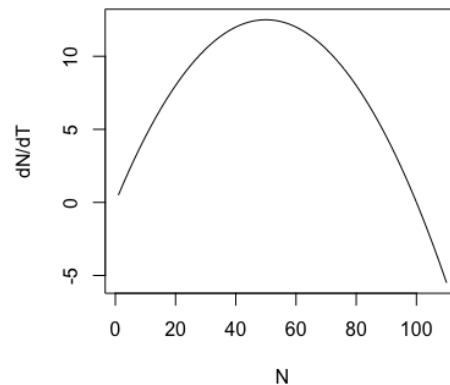
```
logGrowth <- function(t,N){
  # logGrowth gives the growth rate of a population of size N at time t
  r <- .5
  K <- 100
  dNdt <- r*N*(1-N/K)
  return(dNdt)
}
```

Now you can use your function like one of R's, for example:

```
> j <- 1:110
> plot(j,logGrowth(0,j)
      , xlab='N'
      , ylab='dN/dT'
      , type='l'
      )
```

Note that to continue to next line you can press 'shift + enter'.

The code gives an upside down parabola, denoting that growth rate is positive between 0 and 100 and negative when N is greater than 100.



3 Getting Started with Core Techniques

3.1 Create loops

The two types of loops we will discuss are “for” and “while” loops. Both loop start with a keyword such as for or while. The “for” loop allows us to repeat certain commands many times with a “counter” variable. The commands to be repeated are inside . Here is one example:

```
> for(i in 1:4){
  print(i-1)
}
```

You can also increment your counter variable by any real number like

```
> for(i in seq(1,0,-.1)){
  print(i-1)
}
```

Or you can even use a set of arbitrary values:

```
> for(i in c(1,3,5,7)){
  print(i-1)
}
```

Exercise on matrices. Define a random matrix A of size 3-by-3. Use a double “for loop” to calculate the square of the entries in A and store the values in another matrix B. (Hint: type ?matrix and ?runif if you don’t know how to define a random matrix)

The second type of loop is the “while” loop. The “while” loop repeats a sequence of commands as long as some condition is met. For example, given a number n , the following code will return the smallest non-negative integer a such that $2^a \geq n$.

```
> smallexp <- function(n){
+ a <- 0
+ while(2**a < n){
+ a = a+1}
+ return(a)
+ }

> a <- smallexp(4)

> print(a)
[1] 2
```

Note that in the above example we used the conditional statement, $2^a < n$ to decide whether the statement within the while loop should be proceeded. Such conditional statements are also used in “if” statements that are discussed in the next section. The relational operator used here is “<”, which means “less than.” Other relational operators that are available in R include:

```
> greater than
>= greater than or equal
<= less than or equal
== equal
!= not equal
```

Simple conditional statements can be combined by logical operators.

```
&& = AND
|| = OR
```

into compound expressions such as the following:

```
(5 > 1) && (5 == 6)
```

3.2 Make decisions

Now, let’s suppose you want your code to make a decision, and the “if” statement is what you need. The general form in R is as follows:

```
if(expression1){
  statements1
} else if(expression2){
```

```

    statements2
} else {
    statements3
}

```

Exercise: Finish the following pseudo-code that gives the factorial of a positive number n , using the recursive formula $n! = n(n-1)!$:

```

factorial_recur <- function(n){
  if ( # ){
    N = 1
  } else{
    N = #
  }
  return(N)
}

```

3.3 Go fast, i.e., “vectorize”

Remember: for loops are SLOW. One way to make your R code run faster is to “vectorize” the algorithm you use in the code. Vectorization can be done by converting for and while loops to equivalent vector or matrix operations. A simple example would be the following. System.time is a function that captures the runtime of your command.

```

> x<-0; y<-c()
> system.time(
+   for(i in 1:1000){
+     y <- c(y,log10(x))
+     x <- x+0.01
+   }
+ )

> system.time(
+   {
+     x <- seq(0,10,0.01);
+     y <- log10(x)
+   }
+ )

```

For more complicated code, vectorization is not always so obvious. Nonetheless, there is a simple procedure that will help: elementwise operators.

3.4 Elementwise operators

In R, you can do calculations element by element. For example, if you have a variable X and want to square it. In R squaring is denoted by `**` or `^`.

```

> x <- c(5,5,5,5)
> x^2

```

Did it work? Yes! In R elementwise multiplication is the default behavior to interpret the subsequent operator element by element. This is also helpful for a whole matrix. If you would like to do dot multiplication, simply use `%*%` operator.


```
> x <- 5 * matrix(1,4,4)
> x %*% x
```

Contrast the two answers. In the first, you squared all the 5-s. In the second you multiplied two square matrices by each other.

Exercise: Take the square root of the numbers between 1 to 20 using an elementwise operator.

3.5 Find values you want to know about

Given an array X, the command

```
> which(x!=0)
```

returns the indices of all nonzero elements of X. You can also use a logical expression to define X. For example,

```
> which(x>2)
```

returns the indices corresponding to the entries of X that are greater than 2. Notice that X could also be a matrix. You can also have a list of the elements that satisfy the condition.

```
> x[x>2]
```

However, when you want to locate the entries that satisfy more than one logical expressions, the ‘elementwise’ logical operators (& and |) are used in place of ‘short-circuit’ logical operators (&& and ||).

Exercise: Build a 5-by-5 random matrix A using the command

```
> A <- matrix(runif(5*5),5,5)
```

Find the linear indices of entries whose value is smaller than 1/4 and bigger than 1/6.

3.6 Save and load data

Saving and loading objects (variables, arrays, or other forms of data) can be done in a multitude of ways in R, however; for this course we will use the native R format.

```
> array <- c(10,20,30,40)
> save(array, file = "array.RData")
```

and R will save the specified array in the file name. When you want to load all the variables from the file specified by filename, just type

```
> load("array.RData")
```

4 Random numbers

4.1 Rolling dice... again and again

We’ve already used random numbers in this tutorial, without quite understanding the power and potential of this component of R.

To provide some more context, consider what happens when you roll a fair 6-sided die. There are 6 possible events, all equally likely. You might roll a 1, 2, 3, 4, 5 or 6. In living systems, such “chance” events also shape dynamics. For example, replication may have many error correcting mechanisms, but it is not 100% accurate. One might wonder: perhaps the degree of error correction is itself evolvable (hint: yes it is). Other examples of the influence of random change include the motion of small bacteria and viruses as well as the expression of genes. How can we begin to incorporate randomness into models of living systems?

Let’s start with the simplest of the functions:

```
> runif()
```

The command `rand` calls a random number generator. But what kind of random number generator? Run this command a few times and see if you can figure it out... As is apparent the output numbers are all between 0 and 1. By default, `rand` generates random numbers that are *uniformly* spaced between 0 and 1, that is there is an equal chance of getting a 0.0001 or a 0.4983 or 0.9898 or any other number in the interval.

To verify that this works, do the following:

```
> x <- runif(1000)
> hist(x,20,xlab='Random Value',ylab='Histogram')
```

This code snippet first generates 1000 random numbers and stores them in `x`. Next, the `hist` command generates a “histogram” of `x` with 20 equally spaced bins. In this way you can visually check that the numbers are equally spaced.

Here are three challenge problems:

- Generate 1000 random numbers equally spaced between 0 to 5.
- Generate 1000 random numbers equally spaced between 2 to 7.
- Generate 1000 random numbers equally spaced between -5 to 5.

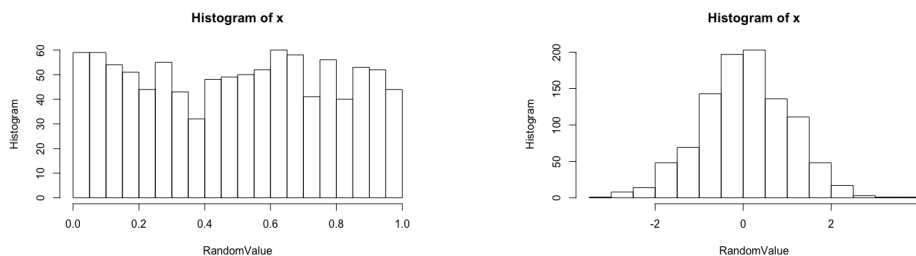
In each of these cases, you should be able to use the command `runif(1000)` and use simple, arithmetic to transform the random numbers. You can do it!

4.2 Normally distributed random numbers... and more

But many processes in biology have very different kinds of randomness. The one you are probably most familiar with is “Gaussian” or “normally” distributed error. R can generate such random numbers. To see how it works, implement this code snippet:

```
> x = rnorm(1000)
> hist(x,20,xlab='RandomValue',ylab='Histogram')
```

The two images below present the contrast in output.



Now that you’ve generated randomly distributed numbers, you might ask: are they really normally distributed? By default, the function `randn` generates random numbers with mean 0 and unit standard deviation.

```
> mean(x)
> sd(x)
```

You will find that your numbers – and each of you may get different numbers – do not have exactly a mean of 0 and a standard deviation of 1, but they are close. You might be worried. I invite you to do the following

```
> library(fitdistrplus)
> FIT <- fitdist(x, "norm")
> plot(FIT)
```

and then discuss with your neighbor to see if the random numbers you sampled could plausibly have arisen from a process that generates randomly distributed numbers with mean 0 and standard deviation 1.

5 Plotting and Reshaping Data

One of the benefits of R is that it is good for manipulating tabular data. In the coming lab, we will be using the "reshape2" package to manipulate the entries of a table.

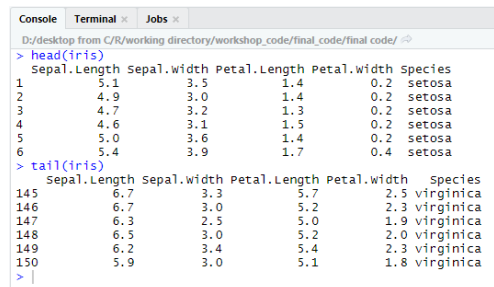
First, install the required package:

```
> install.packages("reshape2")
> require(reshape2)
```

Then, load the following dataset and print the first and last 6 elements :

```
> data("iris")
> iris<-as.data.frame(iris)
> head(iris)
> tail(iris)
```

You should see the following result:

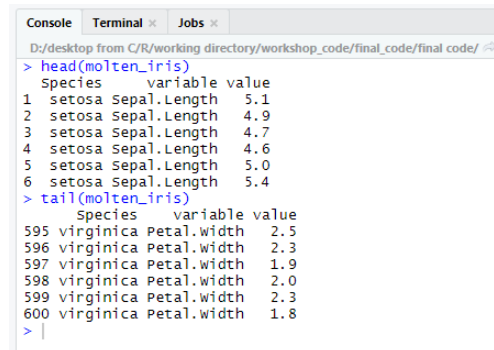


```
Console Terminal x Jobs x
D:/desktop from C/R/working directory/workshop_code/final_code/final code/ ↗
> head(iris)
  Sepal.Length Sepal.width Petal.Length Petal.width Species
1           5.1          3.5          1.4          0.2  setosa
2           4.9          3.0          1.4          0.2  setosa
3           4.7          3.2          1.3          0.2  setosa
4           4.6          3.1          1.5          0.2  setosa
5           5.0          3.6          1.4          0.2  setosa
6           5.4          3.9          1.7          0.4  setosa
> tail(iris)
  Sepal.Length Sepal.width Petal.Length Petal.width Species
145          6.7          3.3          5.7          2.5  virginica
146          6.7          3.0          5.2          2.3  virginica
147          6.3          2.5          5.0          1.9  virginica
148          6.5          3.0          5.2          2.0  virginica
149          6.2          3.4          5.4          2.3  virginica
150          5.9          3.0          5.1          1.8  virginica
> |
```

The "melt" command is used in R to stack data into a "long" format. Try the following commands, in sequence:

```
> melt(iris)
> melt(iris,c('Species'))
> molten_iris<-melt(iris,c('Species'))
> head(molten_iris)
> tail(molten_iris)
```

The last two commands should give the following output:



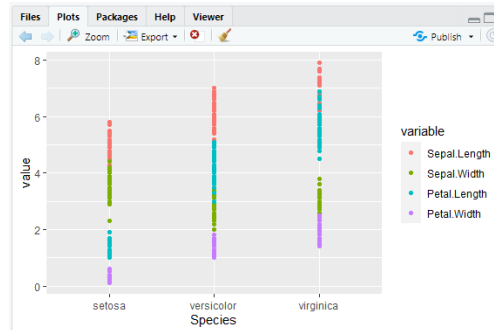
```
Console Terminal x Jobs x
D:/desktop from C/R/working directory/workshop_code/final_code/final code/ ↗
> head(molten_iris)
  Species variable value
1  setosa Sepal.Length  5.1
2  setosa Sepal.Length  4.9
3  setosa Sepal.Length  4.7
4  setosa Sepal.Length  4.6
5  setosa Sepal.Length  5.0
6  setosa Sepal.Length  5.4
> tail(molten_iris)
  Species variable value
595 virginica Petal.width  2.5
596 virginica Petal.width  2.3
597 virginica Petal.width  1.9
598 virginica Petal.width  2.0
599 virginica Petal.width  2.3
600 virginica Petal.width  1.8
> |
```

Hence, we have reshaped the data, arranged by species, into a long stack of variable-value pairs. The argument `c('Species')` identifies the 'Species' column as ID variables to keep in the "molten" dataframe. Additional ID variables can be included. Try to play with variations of these commands.

Next, we would like to visualize this data. The R package "ggplot2" is commonly used for plotting, as shown below:

```
> install.packages("ggplot2")
> require(ggplot2)
> ggplot(molten_iris)+
  geom_point(aes(x=Species,y=value,color=variable))
```

The resulting plot shows points colored according to the category they belong to (sepal and petal lengths and widths):



The x-axis is the value from the 'Species' column of the dataframe (i.e. the three species names).

In the above code, "aes" stands for aesthetic mapping. It is used to describe how variables map onto the plot.

"geom_point" is used to define the type of plot (a scatter plot). There are many other plot type options, such as "geom_line" and "geom_poly".

ggplot has a number of neat features for visualizing data. For instance, the plot can be saved as a variable, and plot-attributes can be added in sequence. For more information, type in the console:

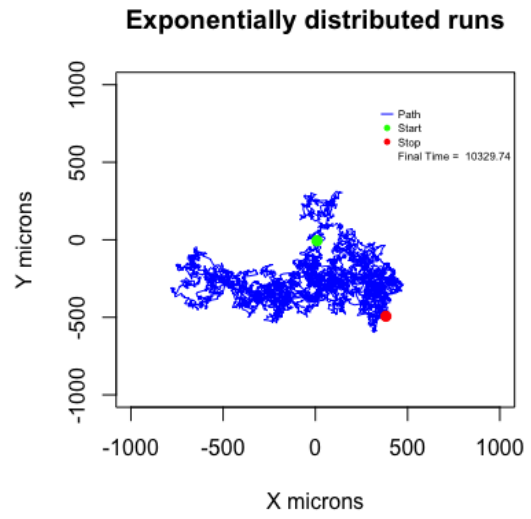
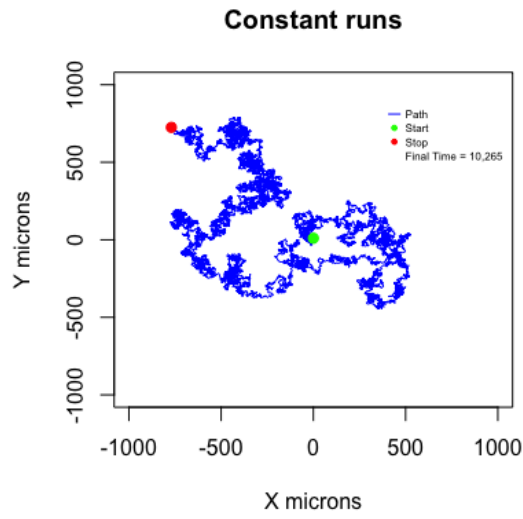
```
> ?ggplot
> ?geom_line
```

6 Advanced - Individual-Based Simulations

For those more experienced in R, try and develop a simulation of "diffusion" in which a bacteria swims at a speed of $10 \mu\text{m/s}$ with each "run" lasting 1 second. In this case the direction of the next run is random. How long will it take, on average, for the bacteria to travel 1mm away from its source? According to the rules of diffusive motion, the average time to travel a distance x should be:

$$T = \frac{x^2}{D} \quad (1)$$

What do we mean by *average* here? If you develop code, can you visualize the results? If you change the length in equation, how does T change? Can you confirm the scaling and estimate D ? What happens if the durations of each run is exponentially distributed, so that runs last on average 1 second, but can vary in duration? Did the answer with respect to travel time to reach 1mm change in a quantitative or qualitative way? We get much further into these ideas in the Foundations of Quantitative Biosciences course. Here are examples of two runs, one with constant duration and one with exponentially distributed durations:



If you are curious for even more, check out these cheat sheets on base R and the ggplot2 package:

<http://github.com/rstudio/cheatsheets/raw/master/base-r.pdf>

<https://github.com/rstudio/cheatsheets/raw/master/data-visualization-2.1.pdf>