

A satellite map of Mexico, showing the country's coastline and inland terrain. The Gulf of Mexico is visible to the east, and the Pacific Ocean to the west. The land is a mix of brown, tan, and green, indicating different geographical features and vegetation.

# Case Studies using the R Programming Language

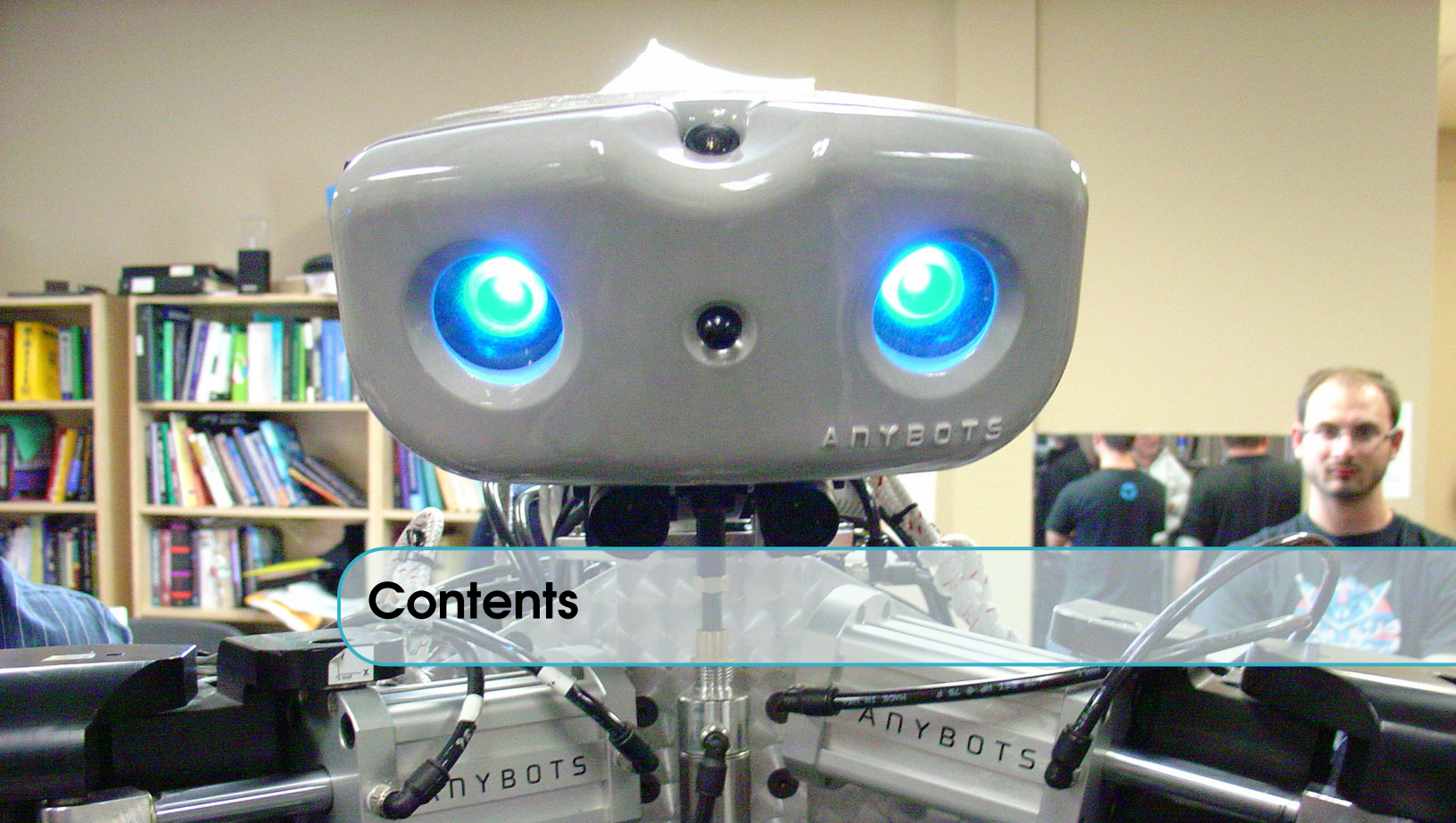
Katherine Bennett, Miguel de los Reyes, Raymond Gao,  
Sophia Hurr, Hannah Gahagan, Nikhil Miland, Mridu Nanda,  
Christa Parrish, Ishaan Rao, Grayson York

RESEARCH IN COMPUTATIONAL SCIENCE

NORTH CAROLINA SCHOOL OF SCIENCE AND MATHEMATICS, DURHAM, NC

This case studies manual was prepared under the direction of Mr. Robert Gotwals, faculty mentor,  
The North Carolina School of Science and Mathematics, Durham NC.

*First release, November 2016*



# Contents

<b>1</b>	<b>ANOVA</b>	<b>7</b>
1.1	Introductory Reading	7
1.2	Objectives	8
1.2.1	ANOVA	8
1.2.2	Visualization	8
1.3	Building the Model	8
1.3.1	Programming Hints	10
1.4	Deliverable	10
1.5	Teaching Code	10
1.6	Example Student Code	11
1.7	Further Readings	12
<b>2</b>	<b>Neural Networks</b>	<b>13</b>
2.1	Introduction to Neural Networks	13
2.1.1	Neural Networks with Sigmoid Neurons	14
2.1.2	Training in a Neural Network	15
2.1.3	Measuring Predictive Error	15
2.2	Installation	16
2.3	Objectives of Case Study	17

<b>2.4</b>	<b>Case Study - Boston Housing Database</b>	<b>17</b>
2.4.1	Loading and Separating the Data	17
2.4.2	Linear Model for Comparison	18
2.4.3	Neural Network Setup and Execution	19
2.4.4	Graphical Representation	20
<b>2.5</b>	<b>Student Assignment - Temperature Dataset</b>	<b>20</b>
<b>2.6</b>	<b>Downloading The Dataset</b>	<b>22</b>
<b>3</b>	<b>Shiny Package Tutorial</b>	<b>23</b>
<b>3.1</b>	<b>Introductory Reading</b>	<b>23</b>
<b>3.2</b>	<b>Objectives</b>	<b>24</b>
<b>3.3</b>	<b>Building the Model</b>	<b>24</b>
<b>3.4</b>	<b>Deliverable</b>	<b>28</b>
3.4.1	Programming Hints	28
<b>3.5</b>	<b>Teaching Code</b>	<b>28</b>
<b>3.6</b>	<b>Example Student Code</b>	<b>30</b>
<b>3.7</b>	<b>Further Readings</b>	<b>31</b>
<b>4</b>	<b>deSolve</b>	<b>33</b>
<b>4.1</b>	<b>Introduction to Computational Differential Solutions</b>	<b>33</b>
4.1.1	Examples of Differential Equations	33
<b>4.2</b>	<b>Installation</b>	<b>34</b>
<b>4.3</b>	<b>Objectives of Case Study</b>	<b>34</b>
<b>4.4</b>	<b>Case Study - Dampened Mass-Spring System</b>	<b>35</b>
4.4.1	Computational Solutions	36
4.4.2	Improving the System	37
<b>4.5</b>	<b>Student Assignment - Modelling of Genetic Processes</b>	<b>39</b>
4.5.1	Constitutive Gene Expression	40
4.5.2	Gene Transcription Regulation	44
4.5.3	Regulation	46
<b>4.6</b>	<b>Conclusion</b>	<b>47</b>
<b>4.7</b>	<b>Acknowledgements</b>	<b>48</b>





# Introduction

Gotwals will write an Intro.





# 1. ANOVA

## 1.1 Introductory Reading

The most common type of linear models are analysis of variance (ANOVA) and linear regression models. R is built to handle linear models, and as such it is easy to work with ANOVA. ANOVA is a statistical method used to compare two or more means. These values can be used to determine whether a significant correlation exists between variables. We will be using one-way ANOVA, which compares the means between the groups of interest, and determines whether those means are significantly different from each other. If one-way ANOVA returns a significant result, there are at least two group means that are significantly different from each other. It is important to note that ANOVA is an omnibus test statistic, meaning that although ANOVA tells the user there is a significant difference in means, it is unable to show which means in particular differ. ANOVA can only be used with certain types of data configurations. To perform ANOVA, data must have a continuous response variable and at least one categorical factor with two or more levels, in lay terms, ANOVA can only be used with numeric values that can be ordered sequentially, with a certain number of possible responses. (i.e. data comparing object's weights, and each new weight is a level) ANOVA is easier to use if data is from approximately normally distributed populations with equal variances between factor levels, however, as R is built to work with statistics, ANOVA procedures will generally work without incident unless one or more of the distributions or variances are highly skewed. A basic understanding of statistics how to use R is recommended before performing ANOVA. The ANOVA user will create and order factors, make box plots, and combine and stack data. [3] The following functions will be useful in using ANOVA.

1. `aov`
2. `as.factor`
3. `ls.str`
4. `data.frame`
5. `stack`

6. TukeyHSD
7. levels

## 1.2 Objectives

In this assignment, we will use ANOVA to analyze data from ELISA HIV Optical Density Readings. ANOVA identifies the causes of variation, and sorts out the corresponding components of variation with associated degrees of freedom. In this case, we are interested in seeing how HIV Optical Density Readings are related to their lot. The type of ANOVA we are using is one-way between groups, as we are comparing one grouping (the optical density readings) to define the groups (lots).

By the end of this lesson, the reader should be able to:

1. Understand the logic behind one-way analysis of variance.
2. Perform one-way analysis of variance in R for any data.
3. Appropriately interpret results of analysis of variance tests.

### 1.2.1 ANOVA

We will look at variances in data using R's ANOVA functions.

### 1.2.2 Visualization

We will also use our fitted models and our ANOVA data to create box plots and line plots. Much of the information gleaned from ANOVA is presented via numeric such as the sum of square, degrees of freedom, and the mean. ANOVA presents the null hypothesis that there is no difference in means of the treatments, and once this hypothesis is proven incorrect, the question arises of how the treatments differ. The post-hoc test also allows the user to find the differences in means, and specifically categorizes the lower and upper means of the data.

Figure 1.1 shows a box plot of the data before running ANOVA or TukeyHSD. Constructing a box plot before analyzing the data may prove helpful in deciding what kind of analysis would be preferable.

## 1.3 Building the Model

Looking at the data set of interest for ANOVA, note that that data may be in numeric form. Use the function `is.numeric` to review the data. If this function prints TRUE, use the function `as.factor` to change data from numeric to factor. In order to make it easier to manipulate data later on, it is recommended that the factored data is renamed.

---

```
1 lot = as.factor(elisa$Lot)
2 summary(lot)
3 levels(as.factor(elisa$Optical))
4 optical = (as.factor(elisa$Optical))
5 levels(as.factor(elisa$Run))
6 run = (as.factor(elisa$Run))
```

---



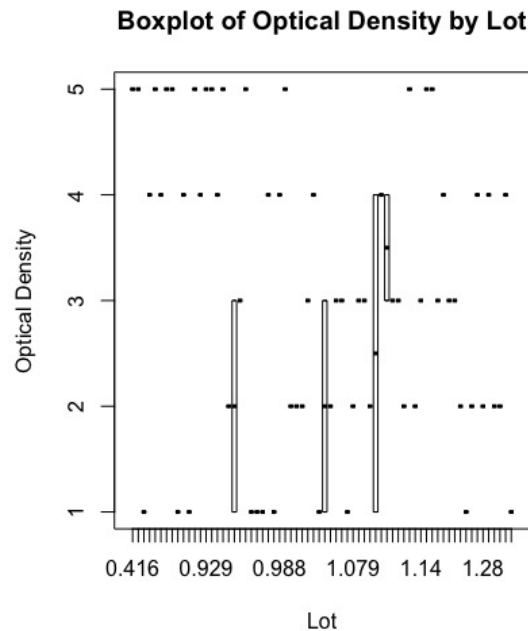


Figure 1.1: Boxplot

Now your factor data type is in non-numerical variables. Each different variable of the factor is called a level. For example, our factor type data for the lots of various ELISA HIV Optical Density Readings has five levels, 1, 2, 3, 4 and 5. Moving on, we can make a box plot of our data, so that we can visually compare data. Making a summary of this data allows us to look at the residuals and standard error of the plot. As mentioned before, this type of comparison will work better with data that has a relatively normal distribution. Factors can also be created within factors. We used our factor Optical to categorize and label by the density. For each level within the main factor, create labels. These labels will designate unordered factors. To make this data easier to read, unordered factors should be ordered. Choose ways to order the data that is relative to the averages in the data. For example, some levels were Low density (below 1) and some were High density, (above 1.5) so we ordered the data starting at 0, then continuing with 1, 1.5, 2. Choose labels that reflect the data, noting that the first label will relate to the lower numbers. Next, classify this newly ordered factor, and now label the data in lay terms.

---

```

1 elisa$Optical.type<-ordered(cut(elisa$Optical, c(0,1,1.5,2), labels = c("Low", "M
2 class(elisa$Optical.type)
3 elisa$Optical.type

```

---

Note, that if you wish to remove one group of subjects (i.e. Lot 1) R will keep this removed group as a level, possibly skewing your data. Use the function drop levels to remove a selected group as a level. Rename this reconfigured data as to not overwrite your previous data. In order to use ANOVA, the data must be in a specific format. To properly configure data, combine the data from the two factors intended to be compared with the function data.frame. Next, stack these combined groups.

Finally use this stacked data and the function `aov` to perform ANOVA. Make a summary of these results.

---

```
1 finalelisa <- droplevels(newelisa)
2 summary(finalelisa$Lot)
3 Combined_Groups <- data.frame(cbind(lot, optical))
4 Combined_Groups
5 Stacked_Groups <- stack(Combined_Groups)
6 Stacked_Groups
7 Anova_Results <- aov(values ~ ind, data = Stacked_Groups)
```

---

This method can be used for a simple one-way ANOVA. However some data can compare multiple sets of data against each other. To do this, we can use a post-hoc test. Post-hoc tests compare outcome measurements between multiple groups. With post-hoc analysis the reader can examine differences between pairs of groups after global analysis. Use the function `TukeyHSD` to perform a pairwise post-hoc analysis on the ANOVA results.

---

```
1 summary(Anova_Results)
2 TukeyHSD(Anova_Results)
3 summary(TukeyHSD(Anova_Results))
```

---

### 1.3.1 Programming Hints

It is recommended to look at data before analysis in order to better understand the levels and attributes of data. In addition, we recommend the use of `summary()` and the console to view the attributes of objects.

## 1.4 Deliverable

Using ANOVA, students should be able to take data with sequentially ordered number data, make a box plot, factor the data as unordered and ordered factors, configure the data such that the function `aov` can be performed, and perform `TukeyHSD`. Using the box plot, students should be able to identify residuals, coefficients of the linear regression, and residual standard error. Using `aov`, students should be able to identify degrees of freedom, the sum of the squares, the mean of the squares, and the F ratio (the ratio of two mean square values) for their data.

## 1.5 Teaching Code

Begin ANOVA by formatting the data in a `.csv` file, then uploading it to RStudio. This script is set up to analyze publicly available data about HIV Optical density readings, but can be adapted for any properly set up data. As the reader works his or her way through the script, be sure to liberally comment the purpose of each command.

---

```
1 # Your name here
2 # Date
3 # ANOVA – ELISA HIV Optical density readings data
4
5 # Clean up and read in data
```

---

---

```

6 rm(list=ls())
7 setwd("C:/Your/Directory")
8 elisa = read.csv("ELISAHIV.csv")
9
10 # Create levels for lot, optical, and run
11 # Make sure to view/check your data
12 levels(as.factor(elisa$Lot))
13 lot=as.factor(elisa$Lot)
14 # etc.
15
16
17 # Create a boxplot showing optical density by lot
18 # Make sure to label your axes!
19 boxplot(..., ...)
20
21
22 # Perform a linear regression of optical density by lot
23 summary(lm(..., data=elisa))
24
25 # Create an ordered factor using Optical.type and verify its type
26 elisa$Optical.type<-ordered(...)
27 class(elisa$Optical.type)
28
29 # Remove 1 as a possible level
30 # Note: even if you eliminate a group of subjects (ex. 1),
31 # because it's a factor, R keeps 1 as a possible level for lot
32
33 # Use droplevels() to remove 1
34
35 # Create stacked results to run aov()
36
37 # Run TukeyHSD on the ANOVA results

```

---

## 1.6 Example Student Code

---

```

1 # KEY
2 # ANOVA – ELISA HIV Optical density readings data
3
4 # Clean up and read in data
5 rm(list=ls())
6 setwd("C:/Example/Code")
7 elisa = read.csv("ELISAHIV.csv")
8
9 # Look at variables and create levels
10 ls.str(elisa)
11 levels(as.factor(elisa$Lot))
12 lot=as.factor(elisa$Lot)
13 summary(lot)
14 levels(as.factor(elisa$Optical))
15 optical = (as.factor(elisa$Optical))
16 levels(as.factor(elisa$Run))
17 run = (as.factor(elisa$Run))
18

```

---

```

19 # Create a boxplot
20 boxplot(elisa$Lot~elisa$Optical, xlab="Lot", ylab="Optical Density",
21         main="Boxplot of Optical Density by Lot")
22 summary(lm(elisa$Optical~elisa$Lot, data=elisa))
23
24 # Perform regression
25 summary(optical)
26 elisa$Optical.type<-ordered(cut(elisa$Optical, c(0,1,1.5,2),
27     labels=c("Low","Medium","High"))))
28 class(elisa$Optical.type)
29 elisa$Optical.type
30
31 # Remove 1 as a level
32 # We've verified that optical is an ordered factor
33 # Note: even if you eliminate a group of subjects (ex. 1),
34 # because it's a factor, R keeps 1 as a possible level for lot
35 newelisa <- elisa[1:2,]
36 summary(newelisa$Lot)
37 # We remove 1 as a possible level using droplevels()
38 finalelisa <-droplevels(newelisa)
39
40 # We create stacked results to run aov()
41 summary(finalelisa$Lot)
42 Combined_Groups <- data.frame(cbind(lot, optical))
43 Combined_Groups
44 Stacked_Groups <- stack(Combined_Groups)
45 Stacked_Groups
46 Anova_Results <- aov(values ~ ind, data = Stacked_Groups)
47 summary(Anova_Results)
48
49 # We also run TukeyHSD on the ANOVA results
50 TukeyHSD(Anova_Results)
51 summary(TukeyHSD(Anova_Results))

```

---

## 1.7 Further Readings

1. Seefeld, Kim and Linder, Ernst. *Statistics Using R with Biological Examples*, University of New Hampshire, Durham, NH Department of Mathematics and Statistics(2007)
2. Julian J. Faraway. *Practical Regression and Anova using R*, University of Michigan, (2002)



## 2. Neural Networks

### 2.1 Introduction to Neural Networks

In the field of computational research, the use of computers is generally restricted to discreet, exact commands that are meant to be executed in a certain manner, which provide the computational power required for complex calculations. In contrast, neural networks are an example of machine learning (ML), wherein the machine is capable of learning behavior or interpreting patterns from a given data set. The field of predictive analysis uses this paradigm of computational interpretation. For example, it is extremely challenging to develop a system that is purely programmed to recognize handwriting. Yet, computers at the United States Postal Service are able to decipher human words and reroute letters and packages without the need for humans. These computers contain a neural network that has been trained to recognize handwriting and predict characters as they appear on mail. [2]

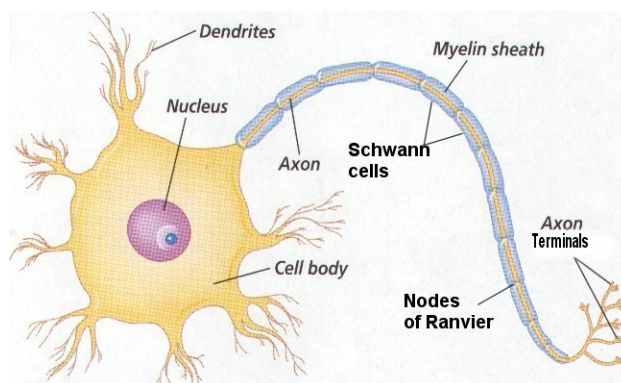


Figure 2.1: The Parts of an Anatomical Neuron

Neural networks were designed using the human brain as an inspiration. The neural network is

composed of perceptrons, which are similar to the smallest functional units of the brain: the neuron. Neurons act as the wiring of the brain by passing along signals that they receive to other neurons. When a neuron communicates with another neuron, it uses neurotransmitters to either increase or decrease the probability of the next neuron firing. Thus, a neuron can be excitatory (increases the chance of the next neuron firing) or inhibitory (decreases the chance of the next neuron firing). Similarly, neural networks utilize perceptrons that can receive multiple inputs and produce a single output. The inputs are binary values (0 or 1), and affect the perceptron differently depending on their excitatory or inhibitory ability. The amount of excitation or inhibition is captured by weights, or values that the inputs are multiplied by. Finally, the aggregate sum of these inputs and their weights is used to calculate if the perceptron in question will fire an output to the next layer of perceptrons.

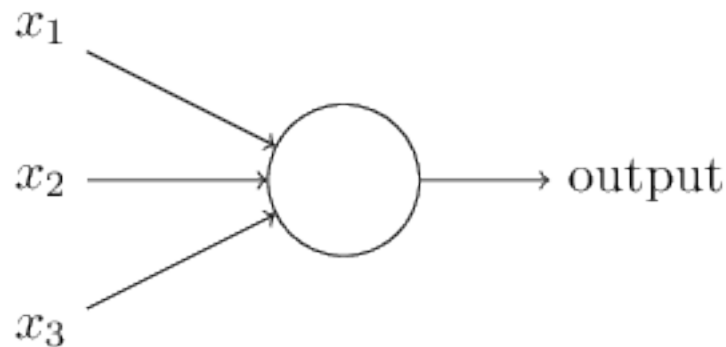


Figure 2.2: A neuron in a neural network, with binary inputs  $x_1$ ,  $x_2$ , and  $x_3$ .

Neurons that only utilize binary inputs, however, are not efficient in learning. For example, neurons in the brain release different amount of neurotransmitters to indicate different levels of excitation. Similarly, most modern neural networks generate an output that is between 0 and 1, inclusive. This type of function is known as the sigmoid function, and the resulting neuron is known as the sigmoid neuron.

### 2.1.1 Neural Networks with Sigmoid Neurons

With practice, humans can arrange sigmoid neurons to generate a functional unit that can take inputs and produce a certain computational output. Functionally, sigmoid neurons act as NAND gates, which are universal to computational procedures. However, the true value of the neural network lies in data and input values that do not have any discernible pattern. In such cases, neural networks are capable of learning patterns by themselves. Therefore, neural networks can be trained with an initial data set, after which they can predict the output values of new information. Although a single sigmoid neurons can be trained, it is not adequate to use such a unit to capture the entirety of complex patterns found in data sets. Rather, a network of sigmoid neurons is created to assess input values and generate proper outputs. Neural networks are most often arranged in layers. The first layer is known as the input layer, whilst the last layer is known as the output layer. All the neural layers in the middle are known as hidden layers. The sample neural network shown in figure 2.3 takes four inputs in the first layer and produces a singular output. Since all the components involved are sigmoid neurons, the value of the output of the neural network can take any value between 0 and 1.

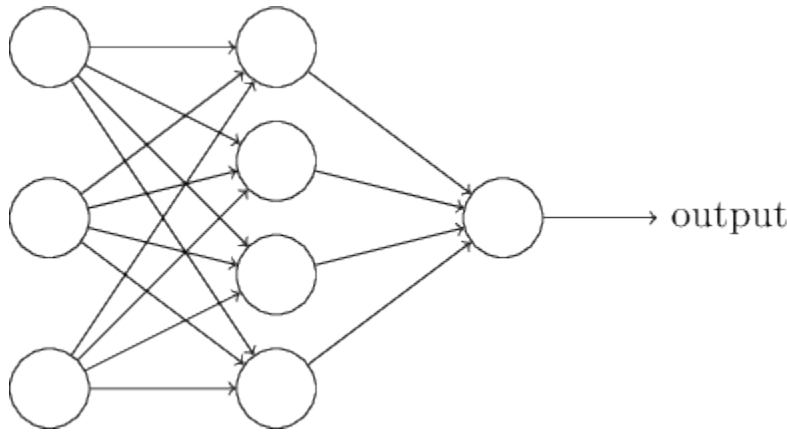


Figure 2.3: A neural network with three layers: an input layer, a hidden layer, and an output layer.

### 2.1.2 Training in a Neural Network

In general, every sigmoid neuron has a certain set of inputs  $X = \{x_1, x_2, \dots, x_n\}$ . Each input is associated with a weight, or the amount of excitation or inhibition each input causes, which are expressed as the weight set  $W = \{w_1, w_2, \dots, w_n\}$ . The value of the previous inputs is also influenced by a neuron-specific bias  $b$ , which acts as a value of how reactive the neuron is to the inputs. Finally, the sigmoid neuron produces an output  $O$  between 0 and 1 depending on the influence of previous inputs as follows:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$O(W, X) = \sigma \left( \sum_i^n w_i x_i + b \right)$$

$$O(W, X) = \sigma(W \cdot X + b)$$

$$O(W, X) = \frac{1}{1 + e^{-W \cdot X - b}}$$

Therefore, by computing the dot product of set  $W$  and set  $X$ , offsetting the dot product by the neural bias, and then passing that value through the sigmoid function, we generate an output between 0 and 1. If the sigmoid neuron in question is part of a hidden layer, it will pass on its output to other neurons in the next layer.

To train a network, we begin with a neural network that is set up with well-chosen (but random) weights and begin sending input values through it. The output is then compared to the value we were expecting, and computational corrections are made in increments to the weights and biases in the neural network using gradient descent algorithms. Over time, the neural network becomes more accurate and has the ability to predict future output values from input values.

### 2.1.3 Measuring Predictive Error

To train our neural network computationally, we need to develop a cost function, that quantifies the amount of training our neural network has undergone. That is, this function has to return a value that

quantifies the difference between prediction and reality. During training, every predicted value is compared to the actual value of the data through subtraction and squared. Thus, for a prediction  $y$  and actual value  $\hat{y}$ .

$$C(y, \hat{y}) = (\hat{y} - y)^2$$

The cost function is computed for every data point used to train the neural network. The overall error of the neural network is simply the average of all the cost function values for every data point that was used to train the network. However, since we are squaring the cost function, we divide the average by 2 to decrease the severity of predictive error. This error function is most commonly known as the Mean-Squared Error function, and is predominantly used to train neural networks. For a set of predictions  $Y = \{y_1, y_2, \dots, y_n\}$  and a set of actual values  $\hat{Y} = \{\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n\}$ :

$$J(Y, \hat{Y}) = \sum_{i=0}^n \frac{C(y_i, \hat{y}_i)}{2n}$$

$$J(Y, \hat{Y}) = \sum_{i=0}^n \frac{(\hat{y}_i - y_i)^2}{2n}$$

## 2.2 Installation

The neuralnet package can be found at <https://cran.r-project.org/web/packages/neuralnet/index.html>. Download the package that works for your operating system and CPU architecture. The following code can be used to install the package from the R command line. This example illustrates the installation of a windows system binary file. However, all other installations will also be the same, with FILE\_PATH leading to the package downloads. Alternatively, a script file can be generated to install the neuralnet package.

Figure 2.4: Code for Windows

---

```
1 install.packages("FILE_PATH.zip", repos=NULL)
2 package 'neuralnet' successfully unpacked with MD5 sums checked
3 library(neuralnet)
```

---

Figure 2.5: Code for Mac / OSX

---

```
1 install.packages("FILE_PATH.tgz")
2 package 'neuralnet' successfully unpacked with MD5 sums checked
3 library(neuralnet)
```

---

Note that safari automatically decompresses .tgz files to tar files, and this will cause errors when running the above code. The installation may throw a warning reminding the user that the



package may be out-of-date when compared to the R version being used.

## 2.3 Objectives of Case Study

There are multiple objectives that the student must accomplish over the course of this neural networks case study.

1. Understand neural networks, their functional units, and their overall purpose in Machine Learning. It is important to understand how neural networks work, beginning from an understanding of sigmoid neurons and culminating in an overall appreciation for the function of neural networks in predictive analysis.
2. Have the ability to prepare data for use in neural networks in R. This objective includes the ability to identify and remove rows with empty data, normalizing the data to allow for learning over a smaller number of iterations, and fitting data curves to create the predictive function.
3. Be able to implement the neuralnet package in R. An understanding of the functionality that the neuralnet package provides, including creating the neural network, training it with a data set, and using it for predictive analysis, is imperative to this objective.
4. Perform cross-validation on the neural network to confirm understand its learning progress. This involves using statistical measures to see how the neural network is functioning during training and prediction.

## 2.4 Case Study - Boston Housing Database

The Boston Housing Database is a database that contains the housing data from the boston area. It contains data on housing sales, including the area of the house sold, the location of the house, and other price-determining factors.

### 2.4.1 Loading and Separating the Data

Create a R script file and enter the following code. This code will set up the environment for the neural network. We load the data into our scripting environment, modify it to fit the neuralnet package, and split it into a training and testing dataset.

---

```
1 # Name:      neural.R
2 # Author:    First Last
3 # Date:      Month Date Year
4
5 # Load necessary libraries into R
6 library(neuralnet) # Contains the neuralnet package
7 library(MASS)      # Contains the database we will use
8
9 # Clear the current environmental variables
10 rm(list=ls())
11
12 # We set the seed for the random generator
13 # We use random data values for training and for testing
14 # Use this seed to replicate the results in this chapter
15 set.seed(500)
```

---

```

16
17 # Import the database into R from the MASS library
18 data <- Boston
19
20 # Define the number of neurons in each hidden layer
21 # The input layer depends on the number of columns in the data frame
22 # The output layer depends on variables we ask the network to predict
23 hiddenLayers = c(5, 3)

```

---

Unfortunately, the neuralnet package cannot handle NULL values in the data. Therefore, we must remove such extraneous data from the dataset.

---

```

1 # Unlike most R libraries , neuralnet is unable to parse NA values
2 # We check for NA values in the data set and print their amount
3 # If we have rows with missing data , we will have to remove them
4 apply(data , 2, function(x) sum(is.na(x)))

```

---

We will use 75% of the data to train the network and 25% to test the efficacy of the neural network. We split the dataset using the following code.

---

```

1 # Generate a list of indices that are randomly sampled
2 # The indices point to rows in the data
3 # 75 percent of the data set indices is selected in this list
4 index <- sample(1:nrow(data), round(0.75 * nrow(data)))
5
6 # The training data set is composed of 75 percent of the test data
7 train <- data[index,]
8
9 # The testing data set is composed of 25 percent of the test data
10 test <- data[-index,]

```

---

### 2.4.2 Linear Model for Comparison

First, we run the data through a simple linear model to see how well a linearly scaled system can predict the value of interest in the data set. Our data frame is constructed so that the last column, named `data$medv`, is what is predicted. Every other column is the value of a pixel in the image.

---

```

1 # Create a regression line using prog as the prognostic variable
2 lm.fit <- glm(medv~., data=train)
3 # Print the summary of our fit
4 summary(lm.fit)
5 # Predict values using the fit and the testing data set
6 lm.pr <- predict(lm.fit, test)
7 # Calculate the Mean-Square Error of the linear model
8 lm.MSE <- sum((lm.pr - test$medv)^2) / nrow(test)

```

---

### 2.4.3 Neural Network Setup and Execution

We will compare the linear model to the neural network graphically after generating results from the neural network. We now work to set up the data for the neural network. Neural networks generally do not work well with non-uniform distributions of data. Therefore, we will scale every column to distribute the values correctly. When we scale the data set, we will lose the true values. Therefore, it is important to "unscale" the data once we generate output from the network.

---

```

1 # Create a list of maximums from each data column
2 maxs <- apply(data, 2, max)
3 # Create a list of minimums from each data column
4 mins <- apply(data, 2, min)
5
6 # Scale the values in each column to its respective limits
7 scaled <- as.data.frame(scale(data, center=mins, scale=maxs - mins))
8
9 # Redefine a training data set using the scaled values
10 train_ <- scaled[index,]
11 # Redefine a testing data set using the scaled values
12 test_ <- scaled[-index,]

```

---

At this point, we can generate a neural network and execute it. We begin by constructing a rule for the neural network to follow. This rule simply describes the output variables and the input variables that are thought to influence the output.

---

```

1 # Every column affects the prognostic variable
2 # We extract the name of every column in the data set
3 n <- names(train_)
4
5 # Using the normal fit construct (prog ~) does not work in this package
6 # Instead, we generate the command using the following code
7 # Will generate something like this: prog ~ c1 + c2 + c3 + ...
8 f <- as.formula(paste("medv ~", paste(n[!n %in% "medv"], collapse=" + ")))
9
10 # The following code will generate the neural network
11 # This step can take some time
12 nn <- neuralnet(f, data=train_, hidden=hiddenLayers, linear.output=T)
13
14 # We can plot the neural network to view its structure
15 plot(nn)

```

---

Just like the linear model that we use to compare, we will calculate the Mean-Squared Error of the neural network. For this, we extract the predictions of the neural network and "unscale" them.

---

```

1 # Compute predictions from our neural network
2 nn.pr_ <- compute(nn, test_[,1:13])
3
4 # Unscale the data output from the neural network
5 nn.pr <- nn.pr_$net.result * (max(data$medv) - min(data$medv)) + min(data$medv)
6 test.r <- (test_$medv) * (max(data$medv) - min(data$medv)) + min(data$medv)
7
8 # Root Mean Square Error computation for the neural network
9 nn.MSE <- sum((test.r - nn.pr)^2) / nrow(test_)

```

---

### 2.4.4 Graphical Representation

Now, we simply use descriptive and graphical methods to highlight the difference between a linear fit model and the neural network.

---

```

1 # A higher number will suggest a lower correlation between the prediction and
  reality
2 print(paste(lm.MSE, nn.MSE))
3
4 # Set up a plot to hold two graphs
5 par(mfrow=c(1, 2))
6
7 # Plot the neural network's real vs. predicted values
8 plot(test$medv, nn.pr, col="red", main="Real vs. Predicted NN", pch=18, cex=0.7,
  xlab="Actual Value", ylab="Neural Network")
9 # Generate a line and legend for the plot
10 abline(0, 1, lwd=2)
11 legend("bottomright", legend="LM", pch=18, col="red", bty="n", cex=0.95)
12
13 # Plot the linear model's real vs. predicted values
14 plot(test$progr, lm.pr, col="blue", main="Real vs. Predicted LM", pch=18, cex
  =0.7, xlab="Actual Value", ylab="Linear Fit")
15 # Generate a line and legend for the plot
16 abline(0, 1, lwd=2)
17 legend("bottomright", legend="LM", pch=18, col="blue", bty="n", cex=0.95)
18
19 # Generate a plot with both scatter plots for better comparison
20 par(mfrow=c(1, 1))
21 plot(test$medv, nn.pr, col="red", main="Real vs. Predicted", pch=18, cex=0.7,
  xlab="Actual Value", ylab="Predicted Value")
22 points(test$medv, lm.pr, col="blue", pch=18, cex=0.7)
23 abline(0, 1, lwd=2)
24 legend("bottomright", legend=c("NN", "LM"), pch=18, col=c("red", "blue"))

```

---

A proper execution of the neural network should show that the Mean-Squared Error of the generalized linear fit model is much higher than that of the neural network. Furthermore, the graphs should portray a tighter clustering of the neural network data points around the linear fit.

## 2.5 Student Assignment - Temperature Dataset

The student is tasked with creating a neural network that interprets a dataset that contains the monthly temperatures from January to November over multiple years. The goal of the network is to predict the temperatures of the month of December. The following code should be used to parse the data:

---

```

1 # Name:          neuralStudent.R
2 # Author:         First Last
3 # Date:           Month Date Year
4
5 ### PARSING ###

```



---

```

6
7 monthlyavg <- c(-4.07, -1.75, 2.17, 8.06, 13.90, 18.77, 21.50, 20.50, 16.27,
8               9.74, 2.79, -2.36)
9 rawdata <- read.csv("berkeleyusdata.txt", header = FALSE)
10 tempdevs <- rawdata$V4
11 temps <- rawdata$V4 + monthlyavg
12 months <- rawdata$V3
13 mat <- matrix(temps, ncol=12)
14 datafornnnet <- data.frame(mat)
15 colnames(datafornnnet) <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",
16                             "Oct", "Nov", "Prog")
17 data <- datafornnnet
18 rm(monthlyavg, rawdata, tempdevs, temps, months, mat, datafornnnet)

```

---

Thus, the data set is stored in a variable called data. The neural network can now be built by the students. The name of the variable being predicted is Prog.

---

```

1 # Clear environment variables
2 rm(list = ls())
3
4 # Import the appropriate libraries
5 library(neuralnet)
6
7 # We do not need to set a seed
8 # We can if the pseudo-random system creates better results
9 # set.seed(500)
10
11 # Vary the number of neurons in the hidden layers to get better results
12 # This is an example a student may use
13 hiddenLayers = c(5, 3)
14
15 # Ensure that the data has no NaN values
16 apply(data, 2, function(x) sum(is.na(x)))
17
18 # Split the data into 75% and 25%
19 # One dataset is for training and the other is for testing
20 index <- sample(1:nrow(data), round(0.75 * nrow(data)))
21 train <- data[index,]
22 test <- data[-index,]
23
24 # Create a generalized linear fit to compare with the neural network
25 lm.fit <- glm(Prog~., data=train)
26 # Print the summary of the fit
27 summary(lm.fit)
28 # Predict values using the fit
29 lm.pr <- predict(lm.fit, test)
30 # Calculate the Mean-Squared Error of the model
31 lm.MSE <- sum((lm.pr - test$Prog)^2) / nrow(test)
32
33 # Calculate mins and maxes for scaling
34 maxs <- apply(data, 2, max)
35 mins <- apply(data, 2, min)
36 # Scale using the range of the data
37 scaled <- as.data.frame(scale(data, center=mins, scale=maxs-mins))
38 # Create the scaled training and test data sets

```

---

```

39 train_ <- scaled[index,]
40 test_ <- scaled[-index,]
41
42 # Create the neural network
43 n <- names(train_)
44 f <- as.formula(paste("Prog ~", paste(n[!n %in% "Prog"], collapse=" + ")))
45 nn <- neuralnet(f, data=train_, hidden=hiddenLayers, liner.output=T)
46 plot(nn)
47
48 # Calculate the scaled predictions and errors
49 nn.pr_ <- compute(nn, test_[,1:11])
50 nn.pr <- nn.pr_$net.result * (max(data$Prog) - min(data$Prog)) + min(data$Prog)
51 test.r <- (test_$Prog) * (max(data$Prog) - min(data$Prog)) + min(data$Prog)
52 nn.MSE <- sum((test.r - nn.pr)^2) / nrow(test_)
53
54 # Print the resulting mean-squared errors
55 print(paste(lm.MSE, nn.MSE))
56
57 # Generate the graphics for displaying data
58 par(mfrow=c(1,2))
59 plot(test$Prog, nn.pr, col="red", main="Real vs. Predicted NN", pch=18, cex=0.7,
      xlab="Actual Value", ylab="Neural Network")
60 abline(0, 1, lwd=2)
61 legend("bottomright", legend="LM", pch=18, col="red", bty="n", cex=0.95)
62 plot(test$Prog, lm.pr, col="blue", main="Real vs. Predicted LM", pch=18, cex
      =0.7, xlab="Actual Value", ylab="Linear Fit")
63 abline(0, 1, lwd=2)
64 legend("bottomright", legend="LM", pch=18, col="blue", bty="n", cex=0.95)
65 par(mfrow=c(1,1))
66 plot(test$Prog, nn.pr, col="red", main="Real vs. Predicted", pch=18, cex=0.7,
      xlab="Actual Value", ylab="Predicted Value")
67 points(test$Prog, lm.pr, col="blue", pch=18, cex=0.7)
68 abline(0, 1, lwd=2)
69 legend("bottomright", legend=c("NN", "LM"), pch=18, col=c("red", "blue"))

```

---

## 2.6 Downloading The Dataset

Download the file located at <https://github.com/nosyarg/textbookdata/blob/master/berkeleyusdata.txt>. This data comes from the Berkeley Earth Project. Once you have downloaded the data, run the following code: as relevant, and make December the prognosis variable

## 3. Shiny Package Tutorial

### 3.1 Introductory Reading

Shiny is a package used in R, that enables the user to easily create interactive applications. Shiny can be used to create both simple histograms of disease data or interactive maps of the crime rate in different cities. With Shiny the possibilities are endless. In this lab, you will learn how to create a basic Shiny application. Look at the picture in Figure 3.1. [4] This is a sample Shiny app depicting a graph of many famous movies. The graph plots movies based on their rating and the number of reviews it has on Rotten Tomatoes. On the left of the graph you find various inputs, such as Oscars and genres, that can customize the output of the graph. These are the basic components of a Shiny app: inputs and outputs. Here are some examples of Shiny input functions:

- `checkboxInput`: a single check box

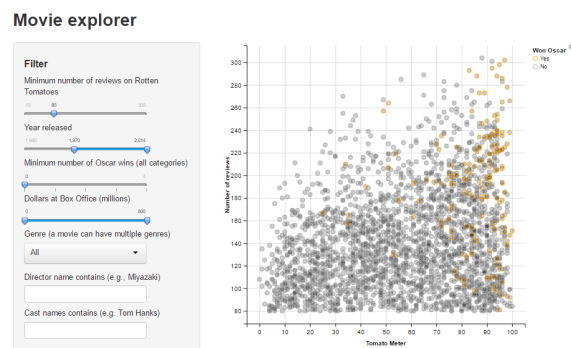


Figure 3.1: Sample Shiny app

- `dateInput`: a calendar
- `fileInput`: file upload
- `numericInput`: enter number
- `radioButtons`: set of radio buttons
- `selectInput`: box with multiple choices
- `sliderInput`: a slider bar
- `textInput`: enter text

The users of a Shiny app will interact solely with the inputs that you, the developer, will allow them to. The above inputs can collect various types of data input from the user, to determine many different types of outputs. Here are some of the output functions in Shiny:

- `imageOutput`: creates image
- `plotOutput`: creates a plot
- `tableOutput`: creates a table
- `textOutput`: creates text

Understanding how inputs and outputs relate to each other is a key requirement for developing a Shiny app. In the model students will create today, the inputs will be dropdown boxes (`selectInput`) and the output will be a simple scatterplot (`plotOutput`).

## 3.2 Objectives

Students will be expected to learn the basic components of a Shiny App. Students will create their own application after learning the fundamentals of the package. This application will illustrate the progression of Olympic swimming medalists' times since the modern Olympics began.

- By the end of this lesson the student will understand the various parts of a Shiny app, while building a simple population prediction application.
- Students will also be able to create their own Shiny apps from scratch, and will be expected to create a Shiny app displaying Olympic swimming times over various events and years.

## 3.3 Building the Model

Shiny applications have a wide variety of uses. Today students will learn how to create a basic population prediction model.

To begin the application, open a new R script file in RStudio, clear the environment, set a working directory, and import the libraries necessary for this project. To clear the environment and set a



directory use the following functions. Make sure you set a directory on your computer that the dataset is located in.

---

```
1 rm(list=ls())
2 setwd("C:/Users/Ishaan/Documents/NCSSM 2015-2016")
```

---

The libraries are as follows:

---

```
1 library(ggplot2)
2 library(scales)
3 library(ggmap)
4 library(dplyr)
5 library(car)
6 library(gcookbook)
7 library(shiny)
```

---

If you don't have one of these libraries installed on your computer, you can simply type

---

```
1 install.packages("insertlibraryname")
```

---

in the console to download them.

Next, we need to import the dataset of all the population data in NC for the next five years into a variable of your choice (I used popData). This can be done with the read .csv function. Make sure the data file is in directory you specified above.

---

```
1 popData <- read.csv("Population.csv")
```

---

Shiny applications have two components: a user-interface definition and a server script. The user-interface is used to control the design and layouts of the Shiny application. The server script for the Shiny application controls the algorithms and code that tell the Shiny app how to manipulate the inputs to produce outputs.[6] Three functions — headerPanel, sidebarPanel, and mainPanel — define the various regions of the user-interface. The application will be called “Population Statistics” so we specify that as the title when we create the header panel. The other panels are empty for now. We create a UI function with the aforementioned components, like this:

---

```
1 ui <- shinyUI(fluidPage(
2   headerPanel("Population Statistics"),
3   sidebarPanel(),
4   mainPanel()
5 ))
```

---

Next, define a skeletal server implementation. This can be done by creating a function called server with two parameters: input and output.

---

```
1 server <- function(input, output) {
2
3 }
```

---

The server function will be empty to begin with, but it will eventually be used to set relationships between the inputs and outputs of the application. At the end of your code, we need to call the shiny function so the application will run. To do that use the following syntax:

---

```
1 shinyApp(ui = ui , server = server)
```

---

The skeletal structure of the simplest Shiny application possible is shown in figure 3.2. Run the Shiny application by selecting all of your code and clicking the run button in the top right corner of RStudio. The blank Shiny application should open in a new window as shown in figure 3.2. To



Figure 3.2: Blank Shiny App

begin the model, we need to display the inputs of our program in the sidebar panel of the application. Our program is taking age group as an input and displaying a plot showing the predicted populations of that age group in North Carolina over the next five years. To do this, we will create a selectInput with all of the age groups as choices, like so:

---

```
1 sidebarPanel(
2   selectInput(
3     inputId = "age",
4     label = "Age:",
5     choices = c(
6       "Total", "0-4 years", "5-9 years",
7       "10-14 years", "15-19 years", "20-24 years",
8       "25-29 years", "30-34 years", "35-39 years",
9       "40-44 years", "45-49 years", "50-54 years",
10      "55-59 years", "60-64 years", "65-69 years",
11      "70-74 years", "75-79 years", "80-84 years",
12      "85+ years")
13   )
14 )
```

---

The inputId is simply a name that we give to each input, so we can access them in the server function. The label is the text that will appear on the application above the input box.

We also need to tell the application that our plot will be displayed in the main panel of the UI. This involves creating a plotOutput, like so:

---

```
1 mainPanel(
2   plotOutput(outputId = "myPlot")
3 )
```

---

Just like in `selectInput`, we need to give the output an ID, so we can use that name to access it in the server function. Once the model is run, the application should look like figure 3.3. Next



Figure 3.3: Adding Inputs

we need to define the server-side of the application which will accept inputs and compute outputs. Our server function will contain the code necessary to do this. We first tell the function that we are creating a plot as an output, and we can create the plot using the `renderPlot` command. The command `renderPlot` is what makes the application react. This means when a user changes the input, the output will instantly change. This is what makes Shiny so interactive.

Within the `renderPlot` function we first select what data we will use and then plot that data using the `ggplot` function. To select our data, create a variable called `dataInput`, and take a subset of the `.csv` file we imported into R. We want to take only the data that the user specified, so instead of importing all the age group data, we only import the age group that the user selected. We can access that specific age group using `input$age`. The `ggplot` function takes in various inputs, such as the data, the variables, and the title. The `aes` function within `ggplot` indicates which axis is given to which variable, while the `geom_point` function draws the points on the graph. The `ggtitle` function creates a title over the plot. The completed server function should look like this:

```

1 server <- function(input, output) {
2   output$myPlot <- renderPlot({
3     dataInput <- subset(popData, (Age==input$age), select=c(Year, Population,
4       State))
5
6     ggplot(
7       data = dataInput,
8       aes(x=Year, y=Population)
9     ) + geom_point() + ggtitle("Population")
10  })
11
12 }
```

Finally, run the entire code including the `shinyApp` command. Your screen should look like the screen displayed in Figure 3.4. The application should have a working select box, a title, and a plot displaying the population data.

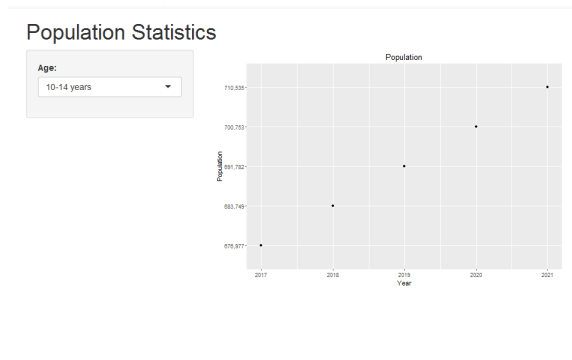


Figure 3.4: Final Model

### 3.4 Deliverable

You are expected to deliver properly formatted code written in the R language, using the Shiny package. Additionally, you will need to provide screenshots of various input combinations to demonstrate that your application is fully functional. This model will display Olympic swimming times over the modern Olympics. The variables for this model include time, year, and medal. The inputs are the event and gender of the swimmer. The graph should display the medaling times for each event and gender for each year of the modern Olympics. The data for this project is found within the `SwimmingExampleM` dataset provided.

#### 3.4.1 Programming Hints

In the deliverable you will need to be able to add color to the model. It is quite simple. In the previous section we explained that the `ggplot` command makes the actual plot. In order to create color we just add `color = variablename`, inside the `aes()` function.

---

```
1 aes(x=Time, y=Year, color=Medal)
```

---

The final shiny app screen should look like figure 3.5

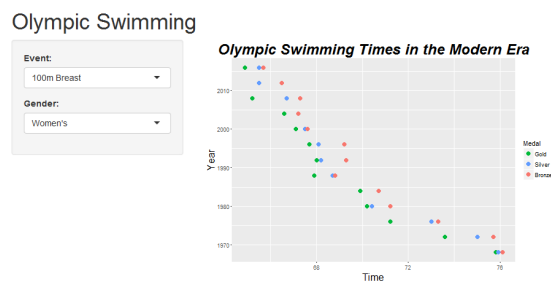


Figure 3.5: Deliverable

### 3.5 Teaching Code

---

```

1 #Ishaan Rao
2 #September 1, 2016
3 #pop.R
4
5 #clean up and set the directory
6 rm(list=ls())
7 setwd("C:/Users/Ishaan/Documents/NCSSM 2015-2016")
8
9 #load libraries
10 library(ggplot2)
11 library(scales)
12 library(ggmap)
13 library(dplyr)
14 library(car)
15 library(gcookbook)
16 library(shiny)
17
18 #read data file
19 popData <- read.csv("Population.csv")
20
21 #setting the server, code instructing R what to do with inputs
22 #output\my plot is what will be created when the app is run
23 #renderPlot allows Shiny to adjust when the user changes an option
24 #By defining the subset we indicate which inputs we want to be selected by the
    user
25 #select are the values in the dataset that are going to be plotted
26 #ggplot give direction to create plot, aes() defines the axes, ggtitle creates
27 title and geom_point creates the scatter plot
28 server <- function(input, output) {
29   output$myPlot <- renderPlot({
30     dataInput <- subset(popData, (Age==input$age), select=c(Year, Population,
        State))
31
32     ggplot(data = dataInput, aes(x=Year, y=Population))
33       + geom_point() + ggtitle("Population")
34
35   })
36
37 }
38 # Define UI for application that draws a scatterplot
39 #headerPanel creates the title
40 #sidebarPanel creates the user entered section
41 #selectInputs is where you insert the options that the user can select
42 #mainPanel outputs the app using all of the previous code we inputed
43 ui <- shinyUI(fluidPage(
44   headerPanel("Population Statistics"),
45   sidebarPanel(
46     selectInput(
47       inputId = "age",
48       label = "Age:",
49       choices = c(
50         "Total", "0-4 years", "5-9 years",
51         "10-14 years", "15-19 years", "20-24 years",
52         "25-29 years", "30-34 years", "35-39 years",
53         "40-44 years", "45-49 years", "50-54 years",
54         "55-59 years", "60-64 years", "65-69 years",

```

---

```

55         "70–74 years", "75–79 years", "80–84 years",
56         "85+ years")
57     )
58 ),
59 mainPanel(
60     plotOutput(outputId = "myPlot")
61 )
62 ))
63 #this runs the Shiny App
64 shinyApp(ui = ui, server = server)

```

---

### 3.6 Example Student Code

---

```

1  #Ishaan Rao
2  #September 1, 2016
3  #olympic.R
4
5  #load libraries
6  library(ggplot2)
7  library(scales)
8  library(ggmap)
9  library(dplyr)
10 library(car)
11 library(gcookbook)
12 library(shiny)
13
14 rm(list=ls())
15 setwd("C:/Users/Ishaan/Documents/NCSSM 2015–2016")
16
17 swimData <- read.csv("SwimExampleM.csv")
18 #setting the server, code instructing R what to do with inputs
19 #output\my plot is what will be created when the app is run
20 #renderPlot allows Shiny to adjust when the user changes an option
21 #By defining the subset we indicate which inputs we want to be selected by the
    user
22 #select are the values in the dataset that are going to be plotted
23 #ggplot give direction to create plot, aes() defines the axes, ggtitle creates
24 title and geom_point creates the scatter plot
25 #scale_color_hue creates different colors for each of the three models\
26 server <- function(input, output) {
27
28     output$myPlot <- renderPlot({
29
30         dataInput <- subset(swimData,
31                             (Event==input$event & Gender==input$gender),
32                             select=c(Time, Year, Medal, Name))
33
34         ggplot(data = dataInput, aes(x=Time, y=Year, color=Medal)) +
35             geom_point(size=3) +
36             ggtitle("Olympic Swimming Times in the Modern Era") +
37             theme(
38                 plot.title = element_text(color="black", size=24,

```



```

39         family="Times", face="bold.italic"), axis.title = element_text(size
           =16)
40     ) +
41     scale_color_hue(breaks=c("Gold","Silver","Bronze"))
42 })
43 }
44 # Define UI for application that draws a scatterplot
45 #headerPanel creates the title
46 #sidebarPanel creates the user entered section
47 #selectInputs is where you insert the options that the user can select
48 #In the model we have two different inputs one for event and one for gender, we
49 create two different selectInput commands to account for the options
50 #mainPanel outputs the app using all of the previous code we inputted
51 ui <- shinyUI(fluidPage(
52   headerPanel("Olympic Swimming"),
53   sidebarPanel(
54     selectInput(
55       inputId = "event",
56       label = "Event:",
57       choices = c(
58         "50m Free", "100m Free", "100m Back", "100m Breast",
59         "100m Butterfly", "200m Free", "200m Back", "200m Breast",
60         "200m Butterfly", "200m IM", "400m IM"
61       ),
62       selected = "50m Free"
63     ),
64     selectInput(
65       inputId = "gender",
66       label = "Gender:",
67       choices = c("Men's", "Women's"),
68       selected = "Women's"
69     )
70   ),
71   mainPanel(
72     plotOutput("myPlot")
73   )
74 ))
75 #runs application
76 shinyApp(ui = ui, server = server)

```

---

### 3.7 Further Readings

- For a full tutorial on creating a Shiny app, visit this tutorial on the Shiny website: <http://shiny.rstudio.com/tutorial/>.
- To learn more advanced Shiny topics, functions, and capabilities visit: <http://shiny.rstudio.com/articles/>.
- To see sample Shiny apps in action visit Shiny's gallery: <http://shiny.rstudio.com/gallery/>





## 4. deSolve

### 4.1 Introduction to Computational Differential Solutions

Many of the systems studied in life sciences, environmental sciences, and physical sciences can be modeled with differential equations. Differential equations serve to model rates of change, which are often easier to measure than every interaction in the system. A differential equation is an equation for a function that relates the values of the function to the values of its derivatives. An ordinary differential equation (ODE) is a differential equation for a function of a single variable (such as  $x(t)$ ), while a partial differential equation (PDE) is a differential equation for a function of several variables (such as  $v(x, y, z, t)$ ). An ODE contains ordinary derivatives and a PDE contains partial derivatives. Typically, PDE's are much harder to solve than ODE's [1].

Initial Value Problems (IVPs) involve differential equations where certain initial values in the model are known. The `deSolve` package uses computational means to analyze and compute the values of states in such models. In a differential equation, the state is the variable that is varied with respect to the independent variable in the system. Parameters, on the other hand, are values that signify some characteristic or attribute of the system. For example, in the given system of equations:

$$\frac{dY}{dt} = aY$$

The state of the differential equation is  $Y$ . The independent variable in this system is  $t$ . Finally,  $a$  is a parameter that can be varied to vary the solution. In this lab, we will analyze systems of ordinary first-order differential equations, which relate the first derivative to states and parameters.

#### 4.1.1 Examples of Differential Equations

The *Malthusian law of population growth* is used to model the populations of certain kinds of organisms living in ideal environments for limited lengths of time  $t$ . It gives the rate at which a

population  $p$  changes with respect to  $t$ . The value of the constant  $r$  depends on the organism.

$$\frac{dp}{dt} = rp$$

This *second-order reaction rate* law gives the rate at which a single chemical species combines to produce a new species, such as methyl radicals combining in a gas to form ethane molecules.

$$\frac{dx}{dt} = k(A - x)^2$$

This equation models the motion of a damped mass-spring system subjected to a time-dependent force  $F(t)$ .

$$m \frac{d^2x}{dt^2} + b \frac{dx}{dt} + kx = F(t)$$

## 4.2 Installation

To install the deSolve package in R, enter the following code into the command line. Installation may produce warnings that are generally inconsequential to the overall usage of the package. However, if the package does not install correctly, contact the package developers.

---

```
1 > install.packages('deSolve')
2 > library(deSolve)
```

---

## 4.3 Objectives of Case Study

There are certain objectives that students should focus to accomplish while completing this activity.

- Know what ordinary differential equations and partial differential equations are. For the case studies that follow, it is inconsequential for students to understand the structure and function of ordinary differential equations.
- Understand how initial value problems arise in ordinary differential equation systems in the natural sciences. That is, use initial values found in nature and differential equation models to produce valid solutions.
- Know how to set up a differential equation given a system and the relationship between rates. This is important for modeling a given system.
- Know how to use deSolve to computationally solve systems of ordinary first-order differential equations. This R package allows for computational solutions that effectively model the changes in the dependent variables of the system.

#### 4.4 Case Study - Dampened Mass-Spring System

As discussed in the examples section, a well-known second-order differential equation is used to analyze the motion of an oscillating mass-spring.

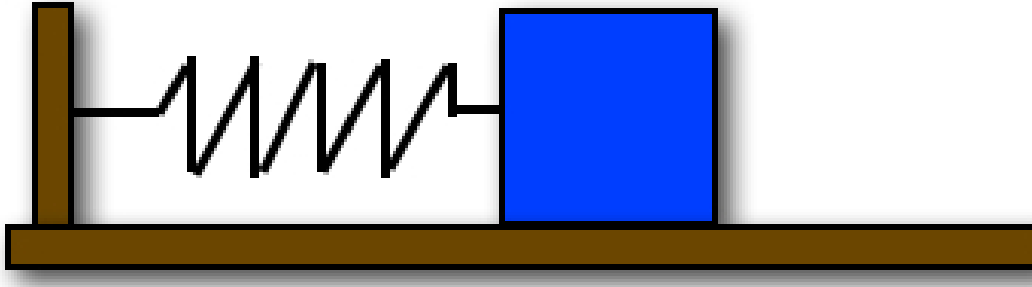


Figure 4.1: A mass-spring system

A mass-spring system is often called a harmonic oscillator. In an idealized system where no heat loss occurs, the mass-spring oscillator will continue oscillating forever.

The nature of a spring can be described using a single parameter, often noted as  $k$ . The spring constant  $k$  is used primarily in Hooke's Law, which relates the amount a spring stretches from rest,  $x$ , to the amount of force  $F$  that is applied on the mass  $m$ . Hooke's Law states that:

$$F = -kx$$

By Newton's Second Law, we also know that:

$$F = m \frac{d^2x}{dt^2}$$

Therefore, we can create a second-order differential equation for the idealized mass-spring oscillator:

$$m \frac{d^2x}{dt^2} = -kx$$

$$\boxed{m \frac{d^2x}{dt^2} + kx = 0}$$

Unfortunately, due to the complexity of behaviors and solutions in second-order differential equations, it is computationally easier to create a system of differential equations  $x'_1, x'_2, \dots, x'_n$  that model the system.

We begin with a few basic states that will represent the differential equations in our system.

$$x_1 = x$$

$$x_2 = \frac{dx}{dt}$$

We compute the derivatives of these states:

$$x_1' = x' = x_2$$

$$x_2' = x''$$

By solving the second-order differential equation in terms of the second derivative  $x''$ :

$$m \frac{d^2x}{dt^2} + kx = 0$$

$$mx'' + kx = 0$$

$$x'' = -\frac{k}{m}x = -\frac{k}{m}x_1$$

Therefore, the system of ordinary first-order differential equations that we are trying to solve is:

$$\frac{dx_1}{dt} = x_2$$

$$\frac{dx_2}{dt} = -\frac{k}{m}x_1$$

#### 4.4.1 Computational Solutions

We begin by setting up the environment to begin using the deSolve package. We import the library and clear the environment variables.

---

```

1 # Author:      First Last
2 # Date:        Month Date Year
3 # Name:        DiffEq.R
4
5 # Import the library into R
6 library(deSolve)
7
8 # Clear variables from environment
9 rm(list = ls())

```

---

As stated in the introduction, we are solving an initial value problem. In this case, we use arbitrary values for states and parameters. We initialize  $x_1 = 100$  to suggest that the spring is stretched 100 meters past its rest phase. Furthermore,  $x_2 = 0$  to suggest that the motion begins from rest, when the rate of change is 0.

---

```

1 # m is the mass of the object (1 kg)
2 # k is the spring constant (50 kg / s^2)

```

---



---

```

3 parameters <- c(m = 1, k = 50)
4
5 # X is the first differential equation (100 m)
6 # Y is the second differential equation (0 m/s)
7 state <- c(X = 100, Y = 0)

```

---

We can define a function that represents the system of differential equations we have defined. Such a function will effectively relate the state and parameters to the derivatives in question.

---

```

1 # Function that defines the system
2 system <- function(t, state, parameters) {
3   with(as.list(c(state, parameters)), {
4     dX <- Y
5     dY <- -1*(k/m)X
6
7     list(c(dX, dY))
8   })
9 }

```

---

We run the simulation over a certain time interval with discrete differences between every time value. Finally, we run the `ode()` function to produce the discrete values.

---

```

1 # The sequence of time run in the simulation
2 times <- seq(0, 10, by=0.01)
3
4 # Generate curves that are solutions to the system of differential equations
5 out <- ode(y=state, times=times, func=system, parms=parameters)

```

---

Running the command above will generate data that is stored in `out`. We can plot this data out. From our derivation, we know that the value of  $x_1 = x$ . Therefore, the solution to  $x_1$  is equal to the solution of the second-order differential equation. We also plot how the two variables,  $x_1$  and  $x_2$ , vary in comparison to each other. We plot the following:

---

```

1 plot(out[, "time"], out[, "X"], xlab="time", ylab="displacement", pch=19)
2 plot(out[, "X"], out[, "Y"], xlab="displacement", ylab="velocity", pch=19)

```

---

#### 4.4.2 Improving the System

However, this idealized system does not capture the motion of a spring that is dampened by other factors, such as heat loss by friction. Such a system is known as a **Dampened Mass-Spring Oscillator**. The spring loses momentum with relation to its velocity, which is the derivative of the position  $x$ . Therefore, we can add a new parameter to the second-order differential equation,  $b$ , that accounts for the dampening of the motion of the spring.

$$m \frac{d^2x}{dt^2} + b \frac{dx}{dt} + kx = 0$$

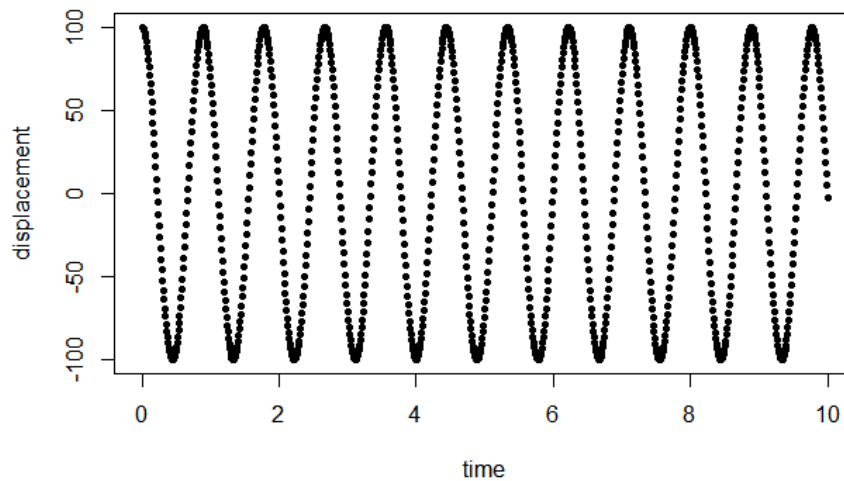


Figure 4.2: The displacement of the spring over time

Using the method described above to convert the second-order differential equation into a system of first-order differential equations, we get:

$$\begin{aligned}\frac{dx_1}{dt} &= x_2 \\ \frac{dx_2}{dt} &= -\frac{k}{m}x_1 - \frac{b}{m}x_2\end{aligned}$$

We modify our systems function in R to account for the velocity-dependent dampening of the spring as follows:

---

```

1 # m is the mass of the object (1 kg)
2 # b is the dampening of the object (0.25)
3 # k is the spring constant (50 kg / s^2)
4 parameters <- c(m = 1, b = 0.25, k = 50)
5
6 # Function that defines the system
7 system <- function(t, state, parameters) {
8   with(as.list(c(state, parameters)), {
9     dX <- Y
10    dY <- -1*(k/m)X - (b/m)Y
11  })
12 }
```

---

Running the simulation again, we get the graphs shown in figures 4.4 and 4.5. Notice that the range of the motion of the spring-mass system continually decreases as time passes. This accounts for much of the heat loss that a dampened spring experiences.

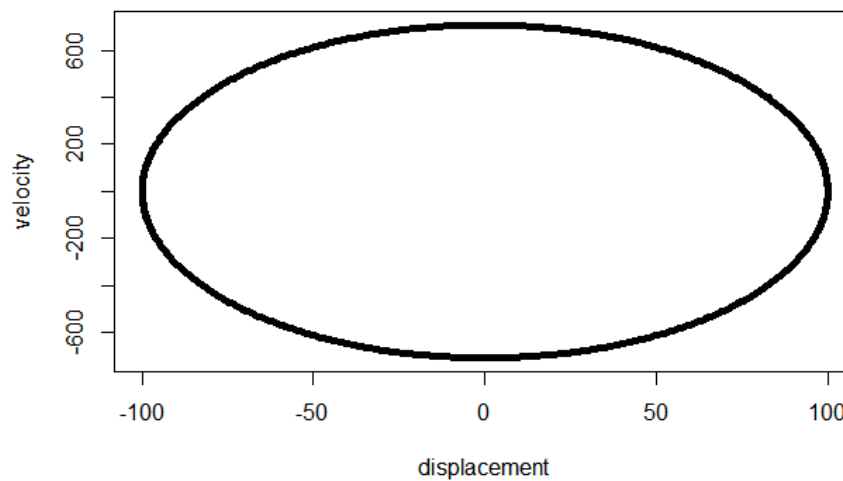


Figure 4.3: The relationship between  $x_1$  and  $x_2$

## 4.5 Student Assignment - Modelling of Genetic Processes

Many of the genetic processes that are studied in the field of molecular biology can be modeled using systems of differential equations. The student is tasked with understanding many of these systems and modeling them. It is important for us to go over the *central dogma of molecular biology* before proceeding any further.

The three main components of this dogma are DNA, mRNA, and proteins. Proteins are generally the effectors in our cells, responsible for life-critical tasks, enzymatic processes, and regulation. Every protein has its own function and specificity. Ribosomal subunits, responsible for synthesizing proteins, get instructions for synthesis from polymer strands made of nucleotides known as mRNA. mRNA strands, in turn, are synthesized using instructions from another polymer known as DNA, which is the only heritable molecule present in the cell (as long as we ignore mitochondrial DNA that is passed down from a maternal lineage). We will also quickly review the processes of transcription and translation in depth. However, a strong foundation in biology is not necessary for modelling the problems that will follow.

- Transcription:** Transcription is the process by which the information contained in a section of DNA (most often a gene) is transferred to a newly assembled piece of messenger RNA (mRNA). Transcription is triggered by the binding of RNA Polymerase to specific regions on the DNA called promoters, and is enabled/facilitated or impeded by a range of promoter-specific proteins called transcription factors [5].
- Translation:** During translation, proteins are produced by ribosomes that bind to a specific site of the mRNA (the ribosome binding site - RBS) and then move along the mRNA chain converting triplets of bases or “codons” on the mRNA into the appropriate peptide chain of amino acids defining the desired protein. The mRNA is read by the ribosome as triplet of bases, usually beginning with an AUG, or an initiator methionine codon downstream

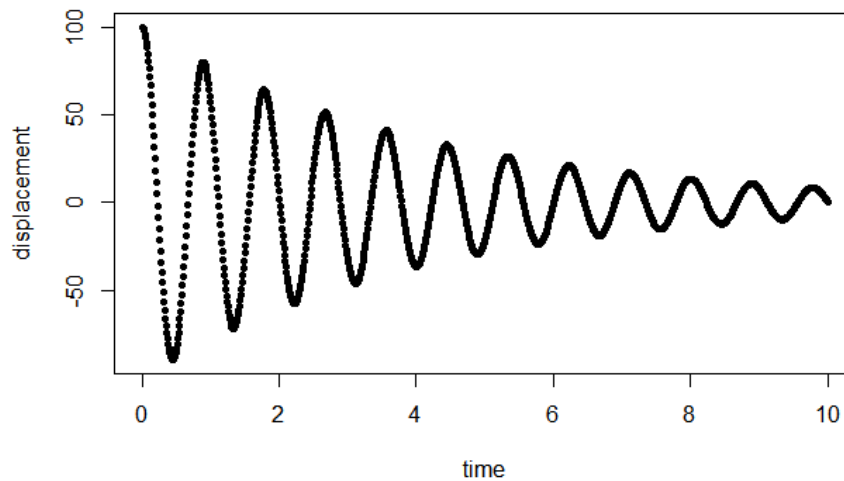


Figure 4.4: The displacement of the spring over time

of the ribosome binding site. Complexes of initiation factors and elongation factors bring aminoacylated transfer RNAs (tRNAs) into the ribosome-mRNA complex, matching the codon in the mRNA to the anti-codon in the tRNA, thereby adding the correct amino acid in the growing peptide sequence encoding the emerging protein. As the amino acids are linked together into the growing peptide chain, they begin folding into the correct conformation. This folding continues until the nascent polypeptide chain is released from the ribosome as a mature protein [5].

#### 4.5.1 Constitutive Gene Expression

**Problem Statement:** Model the system of constitutive gene expression in terms of the central dogma of molecular biology, mRNA degradation, and protein degradation.

Constitutive gene expression is the process that occurs when a gene is always on. That is, the gene is not regulated in any manner and simply follows the central dogma of biology. In such a case, we have to consider a few rates that occur in the system.

- $\frac{dm}{dt}$  is the rate at which mRNA is synthesized from DNA. It will be one of the differential equations students will use in their system.
- $\frac{dp}{dt}$  is the rate at which the protein in question is synthesized from DNA. It will be the other differential equation students will consider in their system.
- $k_m$  is the constitutive transcription rate. It is considered to be constant, and it represents the number of mRNA molecules produced per gene, per unit of time. We assume that there is only one copy of the gene in the cell. In the case of bacterial cells, where plasmid-located genes

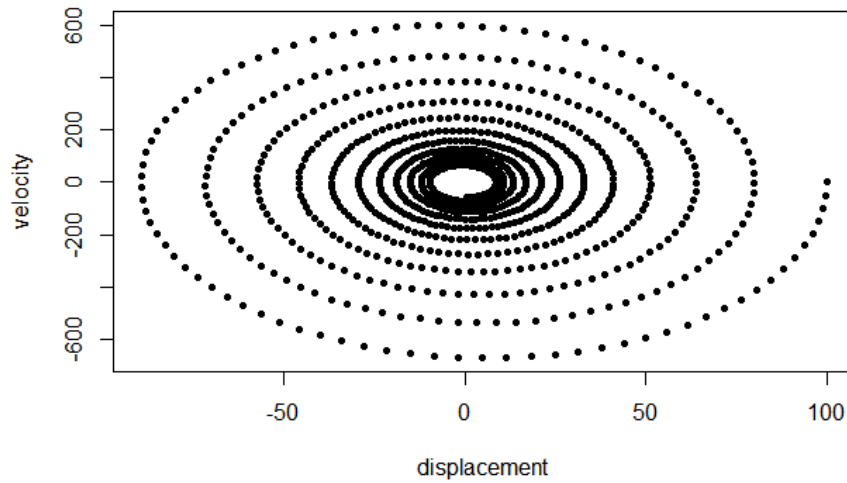


Figure 4.5: The relationship between  $x_1$  and  $x_2$

occur in multiple copies in different plasmids, we would multiply the rate  $k_1$  by the number of copies  $N$  in the cell to obtain a proper rate.

- $k_p$  is the translation rate. It is considered to be constant, and it represents the number of protein molecules produced per mRNA molecule, per unit of time.
- $d_m$  is the mRNA degradation rate, measured as the frequency at which mRNA molecules degrade.
- $d_p$  is the protein degradation rate, measured as the frequency at which protein molecules degrade.

### Sample Student Solution

The student should be capable of modeling the system using the rates provided above. The system we are considering should account for transcription, translation, and the degradation of components in the process.

The rate at which mRNA is transcribed is equal to the rate provided to us,  $k_m$ . Furthermore, we account for the rate of degradation by multiplying the frequency rate  $d_m$  by the number of mRNA molecules in the system. The rate at which protein molecules are translated is equal to the rate  $k_p$ . Similar to the mRNA system, the degradation occurs as a product of  $d_p$  and the amount of protein produced.

$$\begin{aligned}\frac{dm}{dt} &= k_m - d_m m \\ \frac{dp}{dt} &= k_p m - d_p p\end{aligned}$$

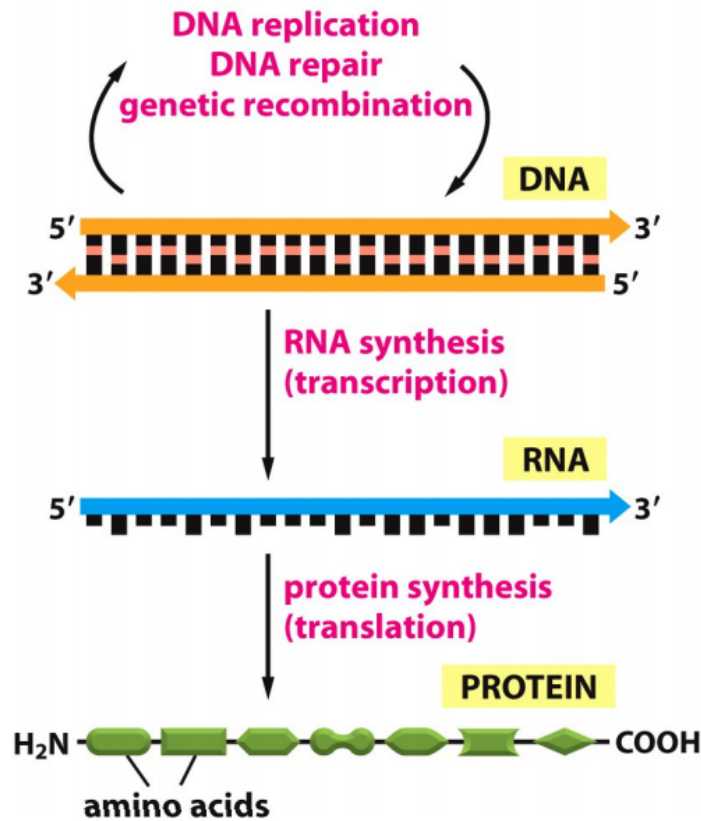


Figure 7-1 Essential Cell Biology 3/e (© Garland Science 2010)

Figure 4.6: The Central Dogma of Molecular Biology

The student simply has to change certain aspects of the program written by the instructor. In this case, the student must define a new set of states, parameters, and a new system function.

```

1 library ( deSolve )
2
3 parameters <- c(km = 10, kp = 5, dm = 1, dp = 3)
4 state <- c(M = 0, P = 0)
5
6 system <- function(t, state, parameters) {
7   with(as.list(c(state, parameters)), {
8     dM <- km - dm*M
9     dP <- dp*M - dp*P
10
11     list(c(dM, dP))
12   })
13 }
14
15 times <- seq(0, 10, by=0.01)
16
17 out <- ode(y = state, times = times, func = system, parms = parameters)
18
19 # Plotting Functions

```



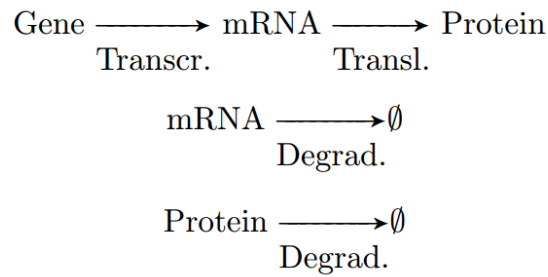


Figure 4.7: The constitutive gene expression system

```

20 xaxis.range <- range(times)
21 yaxis.range <- range(min(range(out[, "M"]), range(out[, "P"])), max(range(out[, "M"]
    ), range(out[, "P"])))
22 plot(out[, "time"], out[, "M"], xlim=xaxis.range, ylim=yaxis.range, col="red",
    xlab="Time", ylab="Concentration", main="System", pch=19)
23 points(out[, "time"], out[, "P"], col="blue", pch=19)
24 legend("bottomright", c("mRNA", "Protein"), pch=c(19, 19), col=c("red", "blue"))

```

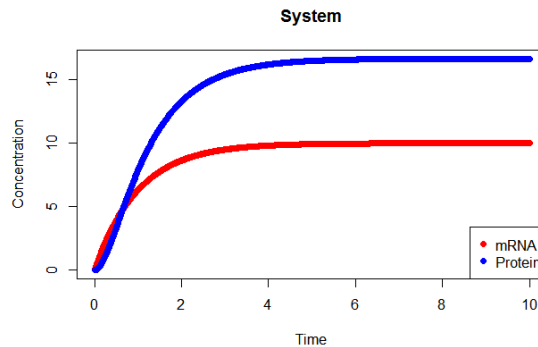


Figure 4.8: Constitutive gene expression concentrations in the system

We see that the concentration of proteins and mRNA in the system reaches a steady state with the initial values supplied. It is important for the student to understand that varying the parameters can change the solution to the system. For example, if the rate of degradation of a protein is much larger than the rate at which it is produced, we expect to see no overall protein production. Therefore, it is just as important to choose appropriate initial values as it is to produce a sound differential model. Although arbitrary values are chosen for these samples, students are encouraged to read literature in their field of research or study to find common values for such problems. For instance, the average half life of mRNA in the model system *E.coli* is around 5 minutes. Such information is inconsequential to producing an appropriate model.

### 4.5.2 Gene Transcription Regulation

Interestingly, most genes that are known today do not follow the constitutive gene expression model. The constant production of proteins can be unnecessary, if not harmful, to the proper function of the organism. Therefore, multiple layers of regulation occur to ensure that the gene produces proteins at the correct time and the proper rate. There are two major arguments for the necessity of gene regulation in cellular and molecular biology. Firstly, all cells carry the same DNA (and therefore, the same heritable information) but have different developmental stages and functions. Secondly, cells are subject to many environmental perturbations, which can only be dealt with by regulation.

Proteins known as transcription factors are generally responsible for the regulation of genes. Transcription factors attach to a region upstream of the gene known as the promoter. The transcription factor can either be an activator (promotes the transcription of the gene) or a repressor (inhibits the transcription of the gene).

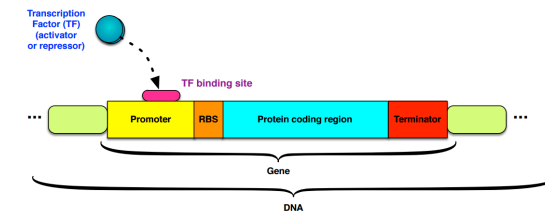


Figure 4.9: Action of regulative transcription factors on protein-encoding DNA regions

#### Activators and Repressors

**Problem Statement:** Model the case of a gene whose transcription is activated by the binding of an activator/repressor to its transcription factor binding site.

A gene begins transcription once an activator binds to its corresponding transcription factor binding site. Once activated, the gene goes through the same process that it would during constitutive gene expression. These processes are inhibited once a repressor binds to the transcription factor binding site. This process as a whole is the basis of gene regulation. Because of the fact that, save for the introduction of activators and repressors, the process that the gene undergoes is identical to constitutive gene expression, many of the rates from the constitutive gene expression section are the same in this case. These include  $\frac{dm}{dt}$ ,  $\frac{dp}{dt}$ ,  $k_m$ ,  $k_p$ ,  $d_m$ , and  $d_p$ . With the introduction of a new parameter (an activator or a repressor), there are subsequently new variables added to this model. The repressor and activator cannot occur at the same time, as the gene can only be activated or inhibited at a certain point in time. These are as follows.

- $A$  represents the activator.
- $R$  represents the repressor.
- $K$  represents the activation or repression coefficient.
- $n$  represents the Hill coefficient, or the number of activators/repressors that need to cooperatively bind to the promoter (transcription factor binding site) in order for gene expression to be activated/inhibited.

**Sample Student Solution**

The student should be capable of modeling the system using the rates provided in the constitutive gene expression section, as well as using the new variables provided above. The system we are considering should account for transcription, translation, and the degradation of components in the process.

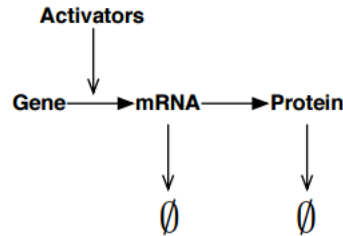


Figure 4.10: The activation and subsequent activities of a gene

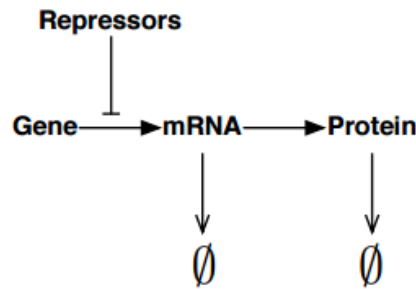


Figure 4.11: The inhibition and subsequent activities of a gene

The differential equations used in this model are almost identical to the differential equations used in previous models (the constitutive gene expression model). The only difference is the introduction of the activator/repressor and its coefficients, which is then multiplied by  $k_m$ .

These are the new differential equations with the introduction of the activator:

$$\frac{dm}{dt} = k_m \frac{A^n}{K^n + A^n} - d_m m$$

$$\frac{dp}{dt} = k_p m - d_p p$$

These are the new differential equations with the introduction of the repressor:

$$\frac{dm}{dt} = k_m \frac{K^n}{K^n + R^n} - d_m m$$

$$\frac{dp}{dt} = k_p m - d_p p$$

As with other cases, the student must define a new set of parameters and system functions in the

code. Most of the code remains the same. The following shows the code when using the activator differential equations.

---

```

1 library (deSolve)
2
3 parameters <- c(km = 10, kp = 5, dm = 4, dp = 4, A = 0.25, K = 0.01, n = 4)
4 state <- c(M = 0, P = 0)
5
6 system <- function(t, state, parameters) {
7   with(as.list(c(state, parameters)), {
8     dM <- km*(A^n/(K^n+A^n)) - dm*M
9     dP <- dp*M - dp*P
10
11     list(c(dM, dP))
12   })
13 }

```

---

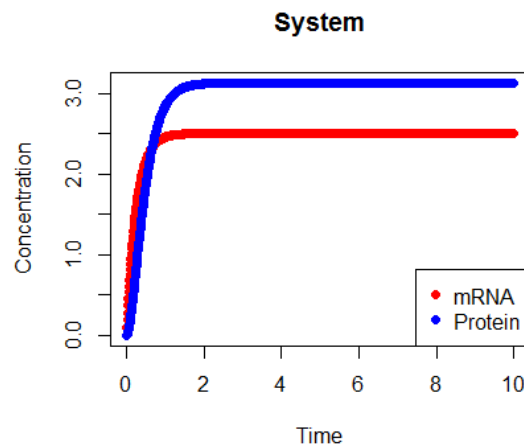


Figure 4.12: Gene expression in a system containing activators

### 4.5.3 Regulation

Unfortunately, by simply solving each system individually, it is hard to tell how activation and inhibition affect gene expression models. With the ability to model constitutive gene expression along with activation and regulation, we can compare models where activation and regulation does occur. Using the following values, we can create three concurrent systems and compare them:

---

```

1 # New parameters in the code
2 parameters <- c(km = 0.346, kp = 6.931, dm = 0.138, dp = 0.017, A = 200, R =
3   200, K = 100, n = 2)
4 state <- c(M = 0, P = 0)
5
6 # Constitutive Gene Expression Model
7 system.const <- function(t, state, parameters) {
8   with(as.list(c(state, parameters)), {

```

---

```

8      dM <- km - dm*M
9      dP <- kp*M - kp*P
10
11      list(c(dM, dP))
12  }
13 }
14
15 # Activation in Gene Expression
16 system.act <- function(t, state, parameters) {
17   with(as.list(c(state, parameters)), {
18     dM <- dm*(A^n / (K^n + A^n)) - dm*M
19     dP <- kp*M - dp*P
20
21     list(c(dM, dP))
22   })
23 }
24
25 # Inhibition in Gene Expression
26 system.rep <- function(t, state, parameters) {
27   with(as.list(c(state, parameters)), {
28     dM <- km*(K^n / (K^n + R^n)) - dm*M
29     dP <- kp*M - dp*P
30
31     list(c(dM, dP))
32   })
33 }
34
35 # Independent variable
36 times <- seq(0, 400, by=1)
37
38 # Computing the solution to all three models
39 out.const <- ode(y = state, times = times, func = system.const, parms =
    parameters)
40 out.act <- ode(y = state, times = times, func = system.act, parms = parameters)
41 out.rep <- ode(y = state, times = times, func = system.rep, parms = parameters)
42
43 # Plotting functions
44 plot(out.const[, "time"], out.const[, "P"], col="red", xlab="Time", ylab="
    Concentration", main="System", pch=19)
45 points(out.act[, "time"], out.act[, "P"], col="green", pch=19)
46 points(out.rep[, "time"], out.rep[, "P"], col="yellow", pch=19)
47 legend("bottomright", c("Protein CONST", "Protein ACT", "Protein REP"), pch=c
    (19, 19, 19), col=c("red", "green", "yellow"))

```

---

The code will produce the graphics shown in figure 4.13. Notice that the constitutive protein expression runs unaffected. The activation gene expression curve takes longer to achieve expression because it takes time for activators to act as transcription factors before transcription can take place. Furthermore, the repressed curve is much lower than both, as is expected from such systems.

## 4.6 Conclusion

Differential equations are used in a variety of different areas to model a wide range of functions, such as the rate that a protein is transcribed or the rate that a population grows. The main function of

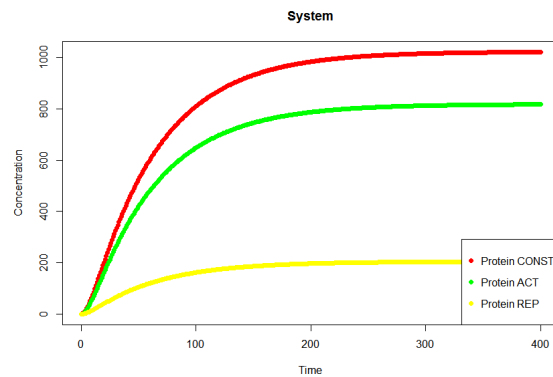


Figure 4.13: Comparing constitutive, activation, and inhibition in gene expression.

a differential equation is to model a rate of change. Some of these equations are easily solved by hand, whereas others are more complicated; the `deSolve` package in R serves to solve and graph a system of differential equation once certain parameters and initial values are entered. Through learning and using `deSolve`, it is possible to easily solve a system of complex differential equations, as well as view the results graphically and be able to change parameters and values as needed. This streamlines a process of solving equations that occur within all types of science, math, and other processes.

## 4.7 Acknowledgements

We would like to thank the creators of the `deSolve` package, Karline Soetaert, Thomas Petzoldt, and R. Woodrow Setzer of the Royal Netherlands Institute of Sea Research. We would also like to thank Mr. Robert Gotwals and the North Carolina School of Science and Mathematics (NCSSM).





## Bibliography

- [1] Jeffrey R. Chasnov. *Introduction to Differential Equations*. 2016. URL: <https://www.math.ust.hk/~machas/differential-equations.pdf> (cited on page 33).
- [2] Michael Nielsen. *Neural Networks and Deep Learning*. Determiation Press, 2015 (cited on page 13).
- [3] Kim Seefeld and Ernst Linder. *Statistics Using R with Biological Examples*. 2007 (cited on page 7).
- [4] *Shiny - Movie Explorer*. Sept. 8, 2016. URL: <http://shiny.rstudio.com/gallery/movie-explorer.html> (cited on page 23).
- [5] Guy-Bart Stan. *Modelling in Biology*. 2010. URL: [http://www.bg.ic.ac.uk/research/g.stan/2010\\_Course\\_MiB\\_article.pdf](http://www.bg.ic.ac.uk/research/g.stan/2010_Course_MiB_article.pdf) (cited on pages 39, 40).
- [6] *Teach Yourself Shiny*. Sept. 8, 2016. URL: <http://shiny.rstudio.com/tutorial/> (cited on page 25).